SOFTWARE REFACTORING USING PATTERN RECOGNITION TECHNIQUES

BY

ABDULAZIZ ALKHALID

A Thesis Presented to the

DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

JANUARY 2009

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **ABDULAZIZ ALKHALID** under the direction of his thesis advisor and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER SCIENCE.**

Thesis Committee

Dr. Sabri Mahmoud (Thesis Advisor)

Dr. Mohammad Alshayeb (Member)

Dr. Mahmoud Elish (Member)

Department Chairman
Dr. Kanaan Faisal

Dean of Graduate Studies
Dr. Salam A. Zummo

24/2/09
Date

DEDICATION

This thesis is dedicated to my parents, my brother and sisters

ACKNOWLEDGEMENT

I would like to begin by thanking Allah, Lord of the worlds, for His mercy. I am asking him always to forgive me and direct me to the right road.

I thank King Fahd University of Petroleum and Minerals for supporting this research. I also thank Dr.Kanaan Faisal the chairman of ICS department and Dr.Adnan Gutub the chairman of COE department.

My deepest appreciation goes to my thesis advisor, Dr.Sabri Mahmoud, who supervised this research. His continual collaboration and strict supervision is highly appreciated. I am very grateful to Dr.Mohammad Alshayeb. His ideas and directions were of great importance to complete this work. I would like to thank the member of my thesis committee: Dr. Mhamoud Elish for his contributions. My thanks go to the Software Engineering Research Group (SERG) for their comments in the weekly meetings. I would like to thank King AbdulAziz City for Science and Technology (KACST) for supporting this research (reference number GSP-17-59).

Furthermore, I would like to thank my friends and colleagues in KFUPM. Finally, I thank the members of my family for their continual prayers. I believe this is what kept me going and gave me the ability to finish my study successfully.

TABLE OF CONTENTS

TABLE OF CONTENTS v
LIST OF TABLES vii
LIST OF FIGURES ix
THESIS ABSTRACT xi
خلاصة الرسالة xii
INTRODUCTION 1
1.1. Introduction 1
1.1. Problem Statement 3
1.2. Main Contribution 3
1.3. Methodology 4
1.4. Organization of the Thesis 5
REFACTORING OF SOFTWARE 6
2.1. Introduction 6
2.2. Software Quality 6
2.2.1. Cohesion and Coupling 7
2.2.2. Measuring Cohesion and Coupling 10
2.2.3. Balancing between Cohesion and Coupling 15
2.3. Refactoring 18
PATTERN RECOGNITION TECHNIQUES 20
3.1. Introduction 20
3.2. Clustering 21
3.3. Classification 23
LITERATURE REVIEW 25
4.1. Introduction 25
4.2. Refactoring at the Function Level 25
4.3. Refactoring at the Architecture Level 28
4.4. Clustering Algorithms 29
4.4.1. Hierarchical Algorithms 30
4.4.2. Optimization and Graph Theoretic Algorithms 31
SOFTWARE REFACTORING AT THE FUNCTION LEVEL 32
5.1. Introduction 32
5.2. Features 33
5.3. Entities and Attributes 33
5.4. Similarity measure 37
5.5. Data Clustering 38
5.5.1 SLINK, CLINK, WPGMA clustering 39
5.5.2. Adaptive K-Nearest Neighbor Clustering (A-KNN) 39
5.6. Refactoring approach using clustering techniques 45
5.7. Experiments on Code Extracted from Published Papers 48
5.7.1. Experimental Results on SLINK, CLINK and WPGMA 49
5.7.2. Experimental Results on SLINK, CLINK, WPGMA and A-KNN 57

5.8. Experimental Results on Industrial System....66
5.9. Conclusions....78
SOFTWARE REFACTORING AT THE CLASS LEVEL....80
6.1. Introduction....80
6.2. Software Refactoring at the Class Level Using Clustering With Fixed Number of Classes....82
6.3. Software Refactoring at the Class Level Using Clustering With Variable Number of Classes....84
6.3.1. The approach....85
6.4. Experimental Results....92
6.4.1. Experimental Results on Software Refactoring at Class Level Using Clustering With Fixed Number of Classes....93
6.4.2. Experimental Results on Software Refactoring at Class Level Using Clustering Without Fixed Cluster-Center....106
6.5. Conclusions....115
SOFTWARE REFACTORING AT THE PACKAGE LEVEL....116
7.1. Introduction....116
7.2. Software Refactoring at the Package Level Using Clustering with Fixed Number of Packages....118
7.3. Software Refactoring at the Package Level Using Clustering With Variable Number of Packages....120
7.3.1. The approach....120
7.4. Experimental Results....123
7.4.1. Experimental Results on Software Refactoring at the Package Level Using Clustering with Fixed Number of Packages....123
7.4.2. Experimental Results on Software Refactoring at the Package Level Using Clustering with Variable Number of Packages....132
7.5. Conclusions....140
CONCLUSIONS AND FUTURE WORK....142
8.1. Introduction....142
8.2. Conclusions and Summary....142
8.3. Future work....146
APPENDIX A: The source code before refactoring....147
APPENDIX B: The source code after refactoring....152
REFERENCES....157
Vita....162

LIST OF TABLES

Table 5.1: Entity-Attribute matrix for the sample code in the Figure 5.1 35
Table 5.2: Attributed matching combinations and their indications 37
Table 5.3: Similarity Matrix of the code in the Figure 5.7 with 2:1 weight ration 51
Table 5.4: Dissimilarity Matrix of the code in the Figure 5.7 with 2:1 weight ratio 51
Table 5.5: Similarity Matrix of the code in the Figure 5.7 with 3:1 weight ratio 53
Table 5.6: Dissimilarity Matrix of the code in the Figure 5.7 with 3:1 weight ratio 54
Table 5.7: Similarity Matrix of the code in the Figure 5.7 with 8:3 weight ratio 55
Table 5.8: Dissimilarity Matrix of the code in the Figure 5.7 with 8:3 weight ratio 56
Table 5.9: Similarity Matrix for the sample code in figure 17 62
Table 5.10: Dissimilarity Matrix for the sample code in figure 17 62
Table 5.11: Output of A-KNN for the sample code in figure 14 with K=3 and 8:3 weight ratio in 12 steps.....65
Table 5.12: Entity-attribute matrix for the source code in the Figure 5.23 69
Table 5.13: Output of A-KNN (K=3) for the source code in the Figure 5.23.....71
Table 6.1: Similarity Matrix for the original source code. 97
Table 6.2: LCOM values before clustering 100
Table 6.3: Values of LCOM after clustering.....100
Table 6.4: Values of CTA before clustering 100
Table 6.5: Change in LCOM and CTA values 100
Table 6.6: Similarity matrix for all classes and methods inside *Environment* package. ..106
Table 6.7: Entity-Attribute Matrix of the system JLOC [63].....108
Table 6.8: Similarity matrix of the system JLOC [63].....109
Table 6.9: CBO and LCOM values of the system JLOC [63] before and after refactoring 114
Table 7.1: Packages and classes in each package of the test source code.....124
Table 7.2: Similarity matrix of the test source code.....124
Table 7.3: New distribution of the classes on the packages of the test source code 124
Table 7.4: Connections among classes of the test source code 125
Table 7.5: Connections among classes of the test source code 125
Table 7.6: Change in the connections number of the test source code after refactoring..125
Table 7.7: Similarity Matrix for the classes and packages in the system Front End For My SQL Domain1.0 [64] before refactoring 128
Table 7.8: Actions suggested by applying the approach 128
Table 7.9: Similarity Matrix for the classes and packages in the system Front End For My SQL Domain1.0 [64] after refactoring 131
Table 7.10: Number of connections of the system Front End for My SQL Domain1.0 [64] before refactoring 131
Table 7.11: Number of connections of the system Front End for My SQL Domain1.0 [64] after refactoring 131

Table 7.12: Change in the connections number of the system Front End for My SQL Domain1.0 [64] after refactoring....132
Table 7.13: Packages and classes inside each package of the system Trama [65]....133
Table 7.14: Number of connections in the original source code of the system Trama [65] 133
Table 7.15: Dissimilarity matrix of the system Trama [65] 134
Table 7.16: Suggested solution by the algorithm SLINK 136
Table 7.17: Number of connections of the system Trama [65] after refactoring using the SLINK algorithm....137
Table 7.18: Suggested solution by the CLINK algorithm....137
Table 7.19: Number of connections of the system Trama [65] after refactoring using the CLINK algorithm 137
Table 7.20: Suggested solution by the WPGMA algorithm....138
Table 7.21: Number of connections of the system Trama [65] after refactoring using the WPGMA algorithm 138
Table 7.22: Suggested solution by the A-KNN algorithm 139
Table 7.23: Number of connections of the system Trama [65] after refactoring using the algorithm A-KNN....140
Table 7.24: Change in the connections number 140

LIST OF FIGURES

Figure 2.1: Coupling among modules 9
Figure 2.2: Example of Lack of cohesion of methods 11
Figure 2.3: Example of lack of cohesion of methods 12
Figure 2.4: Example of coupling through abstract data types 14
Figure 5.1: Sample code 35
Figure 5.2: Example of KNN classification 41
Figure 5.3: Implemented A-KNN, for K=3 44
Figure 5.4: Refactoring Process 45
Figure 5.5: Sample code 46
Figure 5.6: Sample code after preprocessing 47
Figure 5.7: Sample code of reference [56] 50
Figure 5.8: Clustering tree of SLINK with 2:1 weight ratio 52
Figure 5.9: Clustering tree of CLINK with 2:1 weight ratio 52
Figure 5.10: Clustering tree of WPGMA with 2:1 weight ratio 53
Figure 5.11: Clustering tree of SLINK with 3:1 weight ratio 54
Figure 5.12: Clustering tree of CLINK with 3:1 weight ratio 54
Figure 5.13: Clustering tree of WPGMA with 3:1 weight ratio 55
Figure 5.14: Clustering tree of SLINK with 8:3 weight ratio 56
Figure 5.15: Clustering tree of CLINK with 8:3 weight ratio 56
Figure 5.16: Clustering tree of WPGMA with 8:3 weight ratio 57
Figure 5.17: Sample code extracted from reference [25] 58
Figure 5.18: Results expected from refactoring program in Figure 17 61
Figure 5.19: Clustering tree of SLINK for the sample code in figure 17 with 8:3 weight ratio 63
Figure 5.20: Clustering tree of CLINK for the sample code in figure 17 with 8:3 weight ratio 63
Figure 5.21: Clustering tree of WPGMA for the sample code in figure 17 with 8:3 weight ratio 64
Figure 5.22: Clustering tree of A-KNN for the sample code in figure 17 for K=3 with 8:3 weight ratio 65
Figure 5.23: Source code of the function *execute* of reference [58] 68
Figure 5.24: Blocks [1-9], one of the suggested solutions by A-KNN to do refactoring 74
Figure 5.25: Containing class of the method *executes* before refactoring 75
Figure 5.26: Containing class of the method *executes* after refactoring 77
Figure 6.1: Approach to classes refactoring 86
Figure 6.2: Source code before refactoring (a, b, c, d): the source code of class A, B, C, D, respectively 96
Figure 6.3: Source code before refactoring: (a, b, c, d): the source code of class A, B, C, D, respectively 99

Figure 6.4: Coupling between classes before refactoring: (a, b, c, d): coupling between the class A, B, C, D and the other classes, respectively 102
Figure 6.5: Coupling between classes after refactoring: (a, b, c, d): coupling between the class UpdatedA, UpdatedB, UpdatedC, UpdatedD and the other classes, respectively ... 103
Figure 6.6: Clustering Hierarchy of the system JLOC [63] using SLINK algorithm 110
Figure 6.7: Clustering Hierarchy of the system JLOC [63] using CLINK algorithm 111
Figure 6.8: Clustering Hierarchy of the system JLOC [63] using WPGMA algorithm ... 111
Figure 6.9: Clustering Hierarchy of the system JLOC [63] using A-KNN algorithm 113
Figure 7.1: Clustering Hierarchy of the system Trama [65] using SLINK algorithm 135
Figure 7.2: Clustering Hierarchy of the system Trama [65] using CLINK algorithm 135
Figure 7.3: Clustering Hierarchy of the system Trama [65] using WPGMA 136
Figure 7.4: Clustering Hierarchy of the system Trama [65] using SLINK algorithm 139

THESIS ABSTRACT

NAME: Abdulaziz Alkhalid

TITL: SOFTWARE REFACTORING USING PATTERN RECOGNITION TECHNIQUES

MAJOR FIELD: Computer Science

DATE OF DEGREE: Jan, 2009

As the software is enhanced, modified and adapted to new requirements, the code becomes more and more complex. Thus, the quality of the software decreases. Refactoring restructures the code into a more simplified or efficient form in a disciplined way to improve its internal structure without changing external functionality. We investigated refactoring using pattern recognition techniques to balance between cohesion and coupling. The contribution of this thesis is the application of clustering techniques at the function, class, and package levels. In addition, this thesis presents an Adaptive K-Nearest Neighbor (A-KNN) clustering algorithm which was tested at the three levels and compared with Single Linkage, Complete Linkage and Weighted Pair-Group Method using Arithmetic averages algorithms. A-KNN showed its superiority over traditional clustering algorithms in terms of performance and computation complexity. The results were evaluated by comparing them to published work (if available) or by inspection when no published work was available.

خلاصة الرسالة

الاسم: عبد العزيز الخالد

العنوان: إعادة هيكلة البرمجيات باستخدام تقنيات تمييز النموذج

التخصص: علوم الحاسب الآلي

تاريخ التخرج: يناير، 2009

عندما يتم تحسين، تعديل أو تكييف البرمجيات لملائمة المتطلبات الجديدة، يزداد تعقيد الكود البرمجي أكثر فأكثر. وهكذا فإن جودة البرمجيات تتناقص. إن إعادة الهيكلة تعيد تشكيل الكود البرمجي بصورة أكثر تبسيطاً أو فعاليةً و بأسلوب منهجي لتحسين بنيته الداخلية وبدون تغير وظيفته الخارجية. لقد استقصينا إعادة الهيكلة باستخدام تقنيات تمييز النموذج لموازنة التماسك والتجميع. إن إسهام هذه الرسالة هو تطبيق تقنيات العنقدة على مستوى التابع و الصف والحزمة. بالإضافة لما سبق، قدمت هذه الرسالة خوارزمية عنقدة تدعى Adaptive K-Nearest Neighbor (A-KNN). لقد تم اختبار هذه الخوارزمية على المستويات الثلاثة وتمت مقارنتها مع الخوارزميات Single Linkage، Complete Linkage و Weighted Pair-Group Method using Arithmetic averages . لقد أظهرت A-KNN تفوقها على تقنيات العنقدة التقليدية من حيث الأداء و تعقيد الحسابات. لقد قيمت النتائج بالمقارنة مع الأعمال المنشورة (عند توفرها) أو بالتفحص عند عدم توفر عمل منشور.

CHAPTER 1

INTRODUCTION

1.1. Introduction

Software in the real world evolves over time. As the software is enhanced, modified and adapted to new requirements, the code becomes more and more complex. Thus, the quality of the software decreases. The major part of the total software development cost is devoted to software maintenance [1-3]. Thus, there is a need for techniques to reduce software complexity by improving the software quality. This technique is usually referred to as restructuring [4, 5] or, in the specific case of Object Oriented software development, refactoring [6, 7].

Refactoring restructures the code into a more simplified or efficient form in a disciplined way to improve its internal structure without changing external functionality [7]. However, Restructuring can be defined as "*the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior (functionality and semantics). A restructuring transformation*

is often one of appearance, such as altering code to improve its structure in the traditional sense of structured design. While restructuring creates new versions that implement or propose change to the subject system, it does not normally involve modifications because of new requirements. However, it may lead to better observations of the subject system that suggest changes that would improve aspects of the system." [8]. Refactoring, which is an object-oriented variant of restructuring, can be defined as "*the process of changing a [object-oriented] software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure"* [7]. The words "refactoring" and "restructuring" are used interchangeably in the literature. For clarity, however, the present thesis consistently uses the term "refactoring" only.

Enhancing, modifying or adapting the software to new requirements increases the internal software complexity [7, 9]. Software with high level of internal complexity will be difficult to maintain. Consequently, the increase in the maintenance effort will lead to an increase in the overall software cost. Improving the quality of ill-structured programs is one of the most important activities in software engineering. Refactoring is one of the possible solutions to increase the understandability of ill-structured software, hence decreasing the maintenance effort [7]. We used pattern recognition techniques to assist software refactoring activities. These techniques are expected to present computer aided support for identifying ill-structured entities. Furthermore, these techniques were able to present suggestions that make a balance between cohesion and coupling during the software refactoring activities.

1.1. Problem Statement

Software systems evolve over time, and they thus increase in the software internal complexity [7, 9]. After many evolutions, the software understandability decreases because of this increase in the internal complexity [7, 9]. Consequently, software maintenance becomes a time and cost consuming task. One of the solutions to decrease software complexity is software refactoring. Refactoring of software can be done on different levels and for different purposes, which all aim to decrease software internal complexity [7]. In this thesis, we investigate a software metric that can be used for implementing a balance between cohesion and coupling during the software refactoring activities. Then we present a set of approaches, based on pattern recognition techniques, that uses the previous metrics to give suggestions that can enhance the balance between cohesion and coupling among system components.

1.2. Main Contribution

In this thesis, we investigate software refactoring by utilizing a number of pattern recognition techniques. Specifically, we investigate the selection of entities and attributes, software metrics, similarity measures, resemblance coefficient experiments, hierarchical agglomerative algorithms, approaches, and the application of the approaches to an open source system. We provide a set of approaches to identify ill-structured software and give heuristic refactoring advice to software designers in order to improve the balance between cohesion and coupling in evolution phases. The main goal of the presented approaches is

to achieve a balance between software cohesion and coupling based on pattern recognition techniques. The thesis presents a set of techniques supported by pattern recognition mechanisms to specify the parts of software which can be updated and a set of suggestions to achieve a high balance between cohesion and coupling at the level of the software as a whole. Using these techniques, the software designer will be able to get advice which can direct him during the software refactoring activities. In addition to the presented approaches to do software refactoring, this thesis introduces a clustering algorithm called Adaptive K-Nearest Neighbor (A-KNN). The algorithm was tested in many experimental units which showed its superiority over traditional clustering algorithms.

1.3. Methodology

The methodology that we followed in this study includes the following main steps:

1. Identifying the forms of ill-structured software and the entities that can be refactored with the help of pattern recognition techniques.
2. Selecting pattern recognition techniques which can assist in identifying ill-structured entities and give suggest how to balance cohesion and coupling.
3. Adapting pattern recognition techniques to be used in refactoring.
4. Developing approaches to refactor software by using these pattern recognition techniques.
5. Defining the input, output and controllers of these approaches.

6. Applying these approaches on open source systems.
7. Selecting software metrics which can be used to analyze the performance of these approaches.
8. Analyzing the performance of these approaches.
9. Evaluating the performance of these approaches.

1.4. Organization of the Thesis

The rest of the thesis is organized as follows. Chapter 2 discusses the preliminaries of refactoring. Chapter 3 gives the background about pattern recognition required to understand the work presented in this thesis. Chapter 4 provides a brief literature review of the existing work in the thesis area. Chapter 5 explains our work in software refactoring at the function level. Chapter 6 presents our work on software refactoring at the class level. Chapter 7 presents our work in software refactoring at the package level. Finally, chapter 8 presents the conclusions.

CHAPTER 2

REFACTORING OF SOFTWARE

2.1. Introduction

Refactoring of software is intended to enhance the quality of software by improving its understandability, performance, and other quality attributes [7]. This chapter provides an overview of the software quality attributes, specifically cohesion and coupling, and then it presents a general description of software refactoring techniques.

2.2. Software Quality

The American Heritage Dictionary defines quality as “a characteristic or attribute of something”. As an attribute of item, quality refers to measurable properties. Software is more challenging to characterize than physical objects, but measures of a program’s

characteristics do exist. These properties include cyclomatic complexity, cohesion, coupling, number of function points, lines of code, and many others [9].

Two kinds of quality should be taken into account: quality of design and quality of conformance [9]. Quality of design refers to the characteristics that designers specify for a software entity. Quality of conformance is the degree to which the design specifications are followed during software development process.

Making a balance between the software cohesion and coupling is one of the factors that improve software quality through increasing the software understandability. Moreover, software with a high level of understandability will be easier to maintain [9]. Thus, the overall maintenance effort and cost will decrease. However, the balance between software cohesion and coupling becomes a challenging task when the software evolves over time because of the massive increase in the software's internal complexity [9].

2.2.1. Cohesion and Coupling

Cohesion is an internal software attribute that depicts how well the components of a software module are connected. This can be determined by knowing the extent to which the individual components of a module are required to perform the same task [10]. In a highly cohesive module, the component performance is tailored towards the requirement of a single function. On the other hand, a low cohesive module has some elements that

have little relationship with others, which is an indication that the module may provide several unrelated functions [10]. The highly cohesive module is easy to develop and maintain because it does not have much dependence on the components of other modules. This makes it less error-prone.

Coupling is a measure of how strongly one module is connected to, has knowledge of, or relies on other modules [11]. Thus, cohesion addresses intra-module connectedness, while coupling addresses inter-module connectedness.

Coupling and cohesion are closely related. Bad cohesion usually leads to bad coupling because they have a highly interdependent influence [11]. In object-oriented terms, a class with high or strong coupling relies on many other classes. Such reliance may be undesirable for the following reasons [11]:

1) Changes in related classes force local changes.
2) The class is harder to understand in isolation
3) The class is harder to reuse because its use requires the additional presence of the classes on which it is dependent.

Low coupling supports the design of classes or modules that are more independent, which reduces the impact of change. The extreme case of low coupling is not desirable (i.e. when there is no coupling between classes at all or when it is extremely low). If low coupling is taken to excess, it yields a poor design because it leads to a few non-cohesive, bloated, and complex active objects that do all the work [11].

Cohesion, in object-oriented terms, is a measure of how strongly related and focused the responsibilities of a module are. The issue to consider here is how to keep complexity manageable. A class with low cohesion does many unrelated things, or does too much work. Such classes are undesirable; they suffer from the following problems [11]:

1) Hard to comprehend.
2) Hard to reuse.
3) Hard to maintain.
4) Delicate and constantly affected by change.

In general, the relationship between coupling and cohesion is that coupling should be low while cohesion is kept high.

Cohesion and coupling each have different types. Cohesion can be Functional, Layer, Communicational, Sequential, Procedural, Temporal or Utility cohesion. Coupling can be Content, Common, Control, Stamp, Data, Routine Call, Type use, Inclusion/Import, or External coupling. Figure 2.1 shows an example of coupling [9].

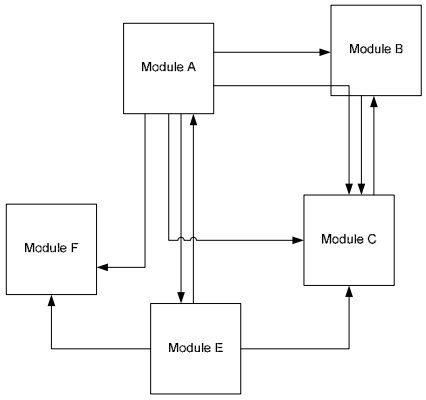


Figure 2.1: Coupling among modules

In Figure 1 each of the modules depends on too many other models in such a way that the designer cannot understand the behavior of one module without understanding the behavior of other modules.

2.2.2. Measuring Cohesion and Coupling

Software designers need metrics to measure the values of cohesion and coupling in order to make decisions which will enhance the software design based on the current values of cohesion and coupling. The Lack of Cohesion in Methods (LCOM) metric can be used to measure cohesion [12], and the Coupling Through Abstract Data Type (CTA) can be used to measure coupling [13].

2.2.2.1. Lack of Cohesion in Methods (LCOM)

Given n methods M_1, M_2, …,M_n contained in a class C_1 which also contains a set of instance variables $\{I_i\}$, then for any method M_i we can define the partitioned set of

$$P = \{(I_i, I_j) \mid I_i \cap I_j = \varphi\} \text{ and } Q = \{(I_i, I_j) \mid I_i \cap I_j \neq \varphi\} \text{ then}$$

$$LCOM = |P| - |Q|, \text{ for } |P| > |Q|$$

$$\text{Otherwise } LCOM=0$$

LCOM is a count of the number of method pairs whose similarity is zero. For example, consider a class C with three methods M_1, M_2 and M_3. Let $\{ I_j\}$ = set of instance variables

used by method M_j. Suppose $\{I_1\}$ = { a, b, c, d, e} and $\{I_2\}$ = {a, b, e} and $\{I_3\}$ = {x, y, z}. The set ($\{I_1\} \cap \{I_2\}$) is nonempty, but ($\{I_1\} \cap \{I_3\}$) and ($\{I_2\} \cap \{I_3\}$) are null sets. LCOM is the (number of null intersections - number of nonempty intersections), which in this case is one. Figure 2.2 shows the methods and their instance variables in the previous example. Figure 2.3 provides more examples for LCOM.

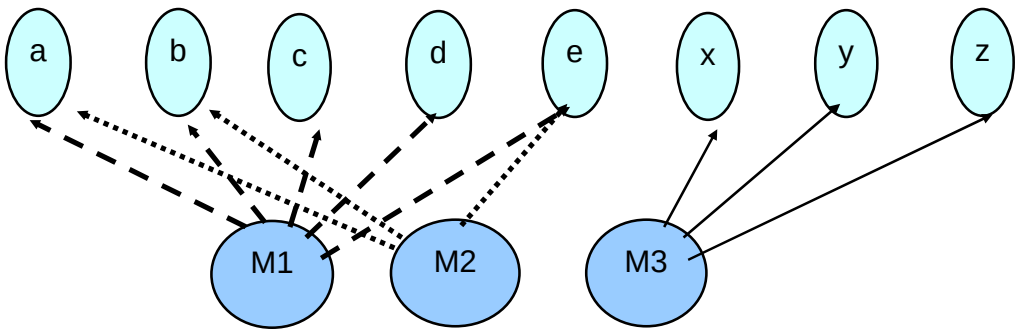


Figure 2.2: Example of Lack of cohesion of methods

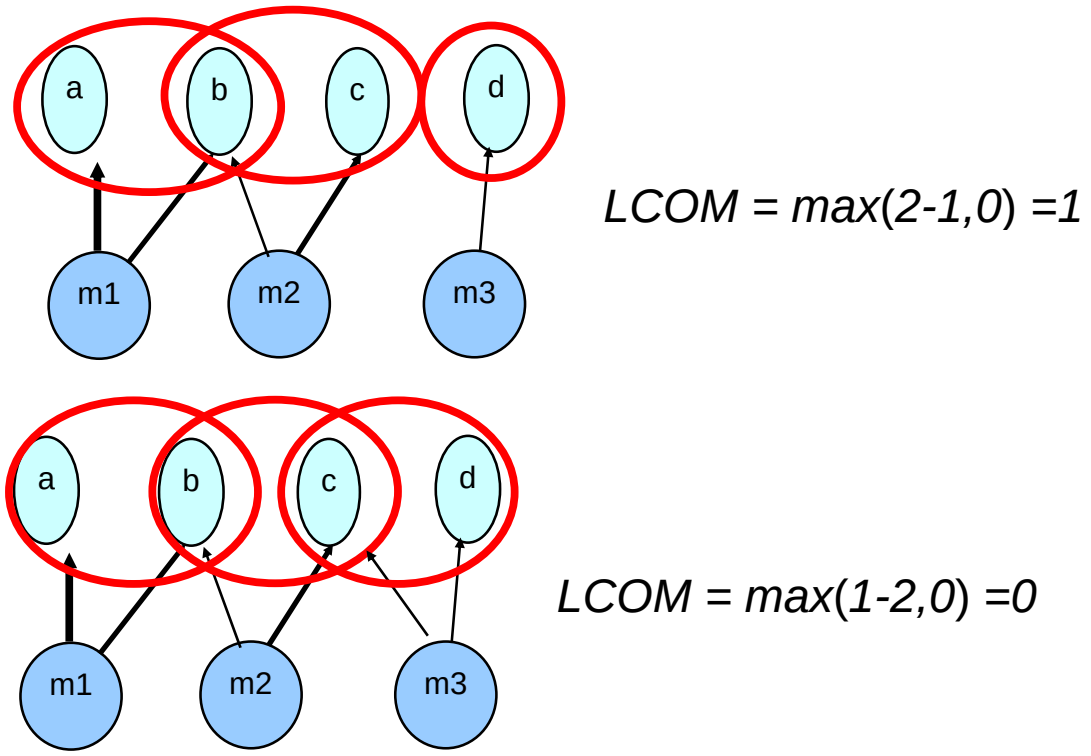


Figure 2.3: Example of lack of cohesion of methods

The theoretical basis for LCOM uses the concept of degree of similarity of methods. The degree of similarity for two methods M_1 and M_2 in class C_1 is given by: $\{I_1\} \cap \{I_2\}$ where $\{I_1\}$ and $\{I_2\}$ are the sets of instance variables used by M_1 and M_2. The LCOM is a count of the number of method pairs whose similarity is zero minus the count of method pairs whose similarity is not zero. Thus, the class is more cohesive if it has a large number of similar methods. Consequently, if all class methods use no instance variables, the LCOM value for the class will be zero. The LCOM depends on the instance variables and methods of a class. In other words, a large number of joint pairs mean a larger similarity of methods.

Cohesiveness of methods within a class is desirable because low cohesion increases complexity. A high LCOM value indicates a disparate class's functionality. Classes that are attempting to achieve many different objectives can be identified by using this metric. Such classes are likely to behave in less predictable ways than classes that have lower LCOM values. Lack of cohesion implies classes should probably be split into two or more subclasses [12].

As a result, the LCOM metric can be used as a way to decide whether the cohesion principle is adhered to in the design of software. Using this metric, software designers can propose changes to enhance software design in the different phases of the software development life cycle.

2.2.2.2. Coupling through Abstract Data Type (CTA)

The Coupling between Object Classes (CBO) metric is defined as "a count of the number of other classes to which it is coupled" [12]. Li claimed that there are different forms of class coupling such as inheritance, abstract data type, and message passing [14]. Thus, it will not be suitable to have one metric for all types of coupling. It is better to have one metric for each type of coupling. In other words, it is better to create a unit definition model for each of the three forms of class coupling than defining one undefined coupling model for the three types.

When a class inherits from another class, directly or indirectly, the two classes are coupled. No metric is proposed to measure this class attribute [13]. Two classes are coupled when one class uses the other class as an abstract data type.

```
class A {                          class B {
        int a;                             A anA;
public:                            public:
        void A();                          void B();
};                                 };
```

Figure 2.4: Example of coupling through abstract data types

Figure 2.4 shows an example where class B is coupled with class A through the use of an abstract data type because class B uses class A in its data-attribute declaration. For this form of class coupling, Li proposed the Coupling Through Abstract Data Type (CTA) metric [13]. However, two classes can be coupled because one class sends a message to an object of another class, without involving the two classes through inheritance or abstract data types. The Coupling Through Message Passing (CTM) metric can be used to measure this type of class coupling [13]. By definition, the CTA metric is the total number of classes that are used as abstract data types in the data attribute declaration of a class. This metric gives the scope of how many other classes' services a class needs in order to provide its own service to others [13].

The unit definition for the CTA metric is “class”. This unit is defined by reference to a standard, which is the use of an abstract data type in OO programming. An analysis of

each data type in a class's data attribute declaration should yield the number of different classes used as abstract data types in the class [13].

2.2.3. Balancing between Cohesion and Coupling

In this thesis we use LCOM as a measure of cohesion and CTA as a measure of coupling. We use CTA as a measure of coupling because we are working on the attribute level. In other words, we are using the attributes to describe the entities which we want to refactor. Thus, CTA will more suitable than CBO and CTM. By using LCOM and CTA we can measure how cohesive is the class and how it is coupled by other classes. According to the measure LCOM and CTA we can suggest some changes that will enhance the balance between cohesion and coupling. For example, consider two classes A and B with the following sample:

```
Class A{
     M1(){
     …
     }
     …
     }

Class B{
     A myA = new A()
     M2() {
          …
          myA.M1();
          …
     }
}
```

In this case, method M1 uses no data members of the class A. So moving the method M1 from class A to any other class will increase the cohesiveness of class A. Moreover, moving the method M1 from class A to Class B will increase the cohesiveness of class A and decrease the coupling of class B. Now, suppose class A has some data members which are used by the method M1.

```
Class A{
     int a, b, c;
     M1(){
          …
          int d=b+c;
          …
     }
     …
}
```

Now, the moving decision is not as easy as it was in the first scenario. Since method M1 uses the data members of class A, then moving it to another class will decrease the cohesiveness of class A, which is not desirable.

```
Class B{
     A myA = new A();
     M1() {
          …
          myA.M1();
          …
     }
     M3(){
          …
          myA.M1();
     }
     M4(){
          …
          myA.M1();
          …
     }
}
```

On the other hand, if the class B has more than one call for method M1(), moving method M1 form class A to class B will decrease the cohesiveness of class A, but it will decrease the coupling of class B with a larger degree. In that case we will have classes like the following:

```
Class A{
     int a, b, c;
     …
}
Class B{
     A myA = new A();
     M1(){
          …
          int d=myA.b + myA.c;
          …
     }

     M2() {
          …
          M1();
          …
     }
     M3(){
          …
          M1();
          …
     }
     M4(){
          …
          M1();
          …
     }
}
```

Thus, moving here is desirable because it will enhance the balance between cohesion and coupling on the level of the system as whole.

Making a balance between cohesion and coupling can be aided by using pattern recognition techniques. Pattern recognition techniques may be used to assist in identifying the current cohesion and coupling of the system. They can also predict among which classes making such a movement will increase the balance between coupling and cohesion on the system level as a whole.

2.3. Refactoring

In the Object Oriented field, software refactoring can be done at different levels. For each level there is a set of refactoring techniques [7, 15]. Class refactorings change the relationships between the classes in the system. Examples of class refactorings can be Add Class, Rename Class, and Remove Class. Method refactorings change the methods within the system. Most of them are analogous to operations that can be performed in most non-object-oriented languages. The added complexity that object-oriented languages bring to these is encapsulation and polymorphism. The list includes Add Method, Rename Method, and Move Method. Variable refactorings work on the level of the variable. The list includes Add Instance Variable, Remove Instance Variable, Rename Instance Variable, Pull Up Instance Variable, Push Down Instance Variable, Abstract Instance Variable, and Move Instance Variable [7, 15].

Adding a new method is allowable if the class and all superclasses of the class do not already define a method with the same name. However, methods with the same name can be added to a subclass as long as it is semantically equivalent to the method defined in the superclass. By adding an equivalent method to a subclass and removing the method from the superclass, we have effectively pushed the method down into the subclass [7, 15].

CHAPTER 3

PATTERN RECOGNITION TECHNIQUES

3.1. Introduction

A pattern is the opposite of chaos; it is an entity vaguely defined, that can be given a name. Recognition is the process of identifying a pattern as a member of a category which is already known. It is the act of taking in raw data and taking an action based on the "category" of the pattern [16].

Different models of classification can be used in pattern recognition systems such as Template Matching, Statistical (geometric), Syntactic (structural), Artificial Neural Network (biologically motivated), and Hybrid approach. These models require different types of features and may have different application areas.

The statistical pattern recognition approach focuses on the statistical properties of the patterns which are generally expressed in probability densities. The model for a pattern

may be a set of features. Thus, the patterns will be represented in a feature space, and the goal is to partition the feature space into categories [16]. Clustering is one of the statistical pattern recognition techniques for organizing the data, and it is one of the possible solutions used to support the refactoring process.

3.2. Clustering

Clustering techniques were originally conceived by Aristotle and Theophrastos in the fourth century B.C. and in the 18th century by Linnaeus, but it was not until 1939 that one of the first comprehensive foundations of these methods was published [17]. Thereafter, many clustering methods were developed [18-20]. Data clustering, as a problem in pattern recognition and statistics, belongs to a class of unsupervised learning. It is a method for clustering data, keeping most similar groups in the same cluster and most dissimilar groups in different clusters [21-23]. Many clustering methods are available, and each of them may give different groupings. However, all of them are related to grouping or segmenting a collection of objects into subsets or clusters, such that those within each cluster are more closely related to one another than objects assigned to different clusters.

Hierarchical and k-means clustering are well known methods of clustering. In hierarchical clustering, the data are not partitioned into particular clusters in a single step. Instead, a series of partitions takes place, which may run from a single cluster containing all objects to several clusters each containing one or more objects [23]. Hierarchical Clustering is

subdivided into agglomerative methods, which proceed by series of fusions of the several objects into groups, and divisive methods, which separate several objects successively into finer groupings. An agglomerative hierarchical clustering procedure produces a series of partitions of the data. The construction of hierarchical agglomerative classification can be achieved by the following general algorithm:

1. Assign each point to a cluster.
2. Find and merge the next two closest clusters, where a cluster is either an individual object or a cluster of objects.
3. If more than one cluster remains, return to step 2.

Individual methods are characterized by the definition used for the identification of the closest pair of points, and by the means used to describe the new cluster when two clusters are merged. There are three basic methods: single linkage clustering, complete linkage clustering and average linkage clustering [23].

In single linkage clustering, the distance between two groups is defined as the distance between the closest pair of objects, taking one object from each group. Here the distance between every possible pair of objects is computed. The minimum value of these distances (called the nearest neighbor) is said to be the distance between the two clusters. In other words, the distance between two clusters is given by the value of the shortest link between the clusters.

The complete linkage clustering is the opposite of single linkage. Distance between groups is now defined as the distance between the most distant pair of objects, one from each group. Here the distance between every possible object pair is computed, and the maximum value of these distances (called the farthest neighbor) is said to be the distance between two clusters. In other words, the distance between two clusters is given by the value of the longest link between the clusters.

In average linkage clustering, the distance between two clusters is defined as the average of the distances between all pairs of objects, where each pair is made up of one object from each group. The distance between two clusters = the sum of all pair-wise distances between the two clusters / (the size of the first cluster * the size of the second cluster).

3.3. Classification

In our work, we will use pattern recognition techniques to aid in program refactoring. One of the possible solutions is using numerical taxonomy or agglomerative hierarchical approaches. The approaches comprise the following common key steps:

1) Obtain the data set.
2) Compute the resemblance coefficients for the data set.
3) Process the clustering assignment.
4) Repeat steps 2 and 3 until no more changes in clustering assignment are produced.

Software code will be parsed to collect some measurements. These measurements will form an input data set which is called the component–attribute data matrix. Components are the entities that we want to group by their similarities. Attributes are the properties of the components. The components and attributes can be many things in various areas, to which clustering analysis can be applied.

A resemblance coefficient for a given pair of components indicates the degree of similarity or dissimilarity between these two components, depending on the way in which the data is represented. For example, the data may be represented by means of a binary variable. If the value of this variable is one, then the component has the property. However, if the data represents a misfit, then the previous value will stand for one possible kind of misfit or dissimilarity. A resemblance coefficient may be qualitative or quantitative. A qualitative value is a binary representation. In that case, the value is either 0 or 1. However, fuzzy logic may be used in this regard to reflect the uncertainty or the degrees of truth that the component has the property. A quantitative coefficient measures the literal distance between two components when they are viewed as points in a two-dimensional array formed by the input attributes.

In summary, there are two types of algorithms for calculating resemblance coefficients based on the scales of measurement used for the attributes. One type of resemblance coefficient can be computed on the basis of qualitative input data or nominal scales for attributes; the other is based on quantitative input data or the attributes that are measured on ordinal, interval, or ratio scales.

CHAPTER 4

LITERATURE REVIEW

4.1. Introduction

There have been extensive studies on software refactoring. This chapter describes related research on refactoring at the function and the architecture level. Furthermore, this chapter presents related research on software clustering.

4.2. Refactoring at the Function Level

Most previous studies of refactoring efforts are focused on making a program's control flow easier to follow [24]. Lakhotia and Deprez [25] used program slicing or input/output dependence techniques to restructure modules with cohesion as the main criterion at the function level. They presented a methodology for refactoring functions with low cohesion into functions with high cohesion, which is a desirable activity during the re-architecting

of a legacy system into an object-oriented architecture. The restructured system has functions with higher cohesion which enables finer-grained grouping of functions into objects. Lakhotia and Deprez presented a technique which partitions the set of output variables of a function on the basis of their pair-wise cohesion. Then program slicing was used to identify the statements that perform computations for each variable group in the partition. New functions corresponding to the slices were created to replace the original function. They conducted a set of experiments with a refactoring real-world code using a tool that implemented the technique.

Kim and Kwon [26] presented methods of refactoring an ill-structured module in the software maintenance phase. Their methods identified modules performing multiple functions and restructure such modules. They achieved this by the notion of the tightly-coupled module that performs a single specific function. This method utilized information on data and control dependence, and applied program slicing to carry out the task of extracting the tightly-coupled modules from the multi-function module. The identified multi-function module was restructured into a number of functional strength modules or an informational strength module. Kim and Kwon used the module strength as a criterion to decide how to restructure. Their proposed methods can be readily automated and incorporated in a software tool.

Kang and Beiman [27, 28] introduced a method to restructure modules during the design or maintenance phases. They claimed that during maintenance phases software developers often use intuition, rather than an objective set of criteria, to determine the design

structure of a software system. A decision process based on intuition alone can miss alternative design options that are easier to implement, test, maintain, and reuse. Kang and Beiman concluded that visual support can supplement human intuition as an ordinal design-level cohesion measure provides objective criteria for comparing alternative design structures. They provided an automated process for visualizing and quantifying design-level cohesion to re-engineer software. Later, they introduced a quantitative framework for software restructuring, in which the restructuring decisions were guided by visualized design information and objective criteria. They extracted design information from the code to refactor existing or legacy software. Consequently, they accomplished refactoring through a series of decomposition and composition operations which increased the cohesion and/or decreased the coupling of individual system components. Their framework ensured that refactoring resulted in measurable improvements in design quality.

Lakhotia and Deprez [25 and 29] used a transformation called Tuck to restructure programs by breaking large functions into small functions. Tuck consists of three steps: Wedge, Split, and Fold. A wedge is a subset of statements in a slice that contains related computations and may create a meaningful function. The method complements those reported in [26, 27] by computing pair-wise cohesion. After a wedge is formed, it is split from the rest of the code and folded into a new function.

The methods in [25, 27-29] extracted computations related to output variables. In reality, a function, which has a single output variable, cannot be decomposed further. For

example, handling routines may not be related to output variables. In such cases, the slices of output variables cannot reflect the code fragment related to error handling. Furthermore, in some functions there is only one output variable, but the function involves multiple activities.

Lung and Zaman [30] applied clustering techniques to function refactoring, and they explained how to refactor a low-cohesive function into high-cohesive functions by using simple examples presented in the literature. They treated executable program statements as basic components, or entities, and variables as attributes. They also introduced artificial variables for iterative loops and logical control statements. Lung et al. [31] extended the previous work and defined a new similarity measure, and they compared various weights systematically and applied the technique to industrial software.

4.3. Refactoring at the Architecture Level

Extensive research on software clustering has been conducted at the design or architectural level [24, 25, 27-53]. Tzerpos and Holt [51] surveyed clustering approaches, and they found that classical clustering techniques can be used in the software context and that there is a research potential in the software clustering field. They pointed out that some structure is better than no structure.

Wiggerts [53] provided a general overview of clustering techniques and their applications to system re-modularization, highlighting the benefit of the general theory of clustering analysis. Lakhotia [42] reported a survey on subsystem classification techniques, and he provided a unified framework for entity description and clustering methods, in order to facilitate comparison between various subsystem classification techniques.

Previous software clustering approaches concentrated on software system modularization or re-modularization at the architectural or design level. The entities to be clustered could be functions (routines), global variables (for identifying abstract data types), or files. Researchers used different information or formula to measures the similarity based on different perspectives. Some researchers used similarity measures based on relationships between entities [31, 44, 47, 48]. Other researchers developed similarity measures based on shared features [32, 39, 50], with or without giving weights to the features.

4.4. Clustering Algorithms

Clustering algorithms in the previous studies may be classified into three categories: hierarchical algorithms, optimization algorithms, and graph theoretic algorithms.

4.4.1. Hierarchical Algorithms

Several studies used hierarchical algorithms in software refactoring [30-33, 39, 50]. One of the hierarchical agglomerative algorithms Unweighted Pair-Group Method using Arithmetic Averages (UPGMA) is a commonly used approach [54]. Lung [31] and Lung et al. [43] presented reverse engineering and reengineering experiences for architecture recovery based on UPGMA. Both UPGMA and Weighted Pair-Group Method using Arithmetic Averages (WPGMA) are average linkage clustering algorithms.

UPGMA considers all pair entities in two clusters or cluster size in calculating the average; whereas WPGMA calculates the simple average. In a basic survey in [42] the authors suggested that most researchers prefer the Single Linkage (SLINK) algorithm in subsystem classification. Girard et al. [55] tailored the SLINK algorithm because it generated very large groups that were not useful and led to what is called tuning. These phenomena happen because some clusters grow larger than others, until one cluster contains most of the leaves. Alternatively, in [32] the authors suggested the use of Complete Linkage (CLINK) algorithm based on their experiments for software re-modularization using files. Maqbool and Babri [46] presented a weighted combined linkage algorithm of software clustering to support design recovery. They compared the presented algorithm with some clustering algorithms.

4.4.2. Optimization and Graph Theoretic Algorithms

Although optimization algorithms were rarely used in the previous research, some studies were conducted in this field [44, 45, 47]. Graph theoretic algorithms represented the least category used in software refactoring. Some examples were presented in [37, 48].

CHAPTER 5

SOFTWARE REFACTORING AT THE FUNCTION LEVEL

5.1. Introduction

This chapter presents an approach to software refactoring at the functional level using clustering algorithms. The objective is to provide automated support to identify ill-structured or low-cohesive functions, and to present refactoring suggestions. These suggestions will provide heuristic advice to assist the software designer in software refactoring. The software entities which can be restructured, and the attributes of these entities, are investigated. Comparisons are made among three hierarchical agglomerative algorithms: Single Linkage algorithm (SLINK), Complete Linkage algorithm (CLINK) and Weighted Pair-Group Method using Arithmetic Averages (WPGMA) with different attributes weights is conducted. In addition, this chapter introduces a new adaptive K-Nearest Neighbor (A-KNN) algorithm to organize those entities and attributes. The new technique is compared to the SLINK, CLINK, WPGMA techniques.

The rest of this chapter is organized as follows. Section 2 explains the features. Section 3 addresses the entities and attributes. Section 4 explains the data clustering. Section 5 describes the refactoring approach using clustering techniques. Section 6 explains the experimental results on a published code. Section 8 presents the experimental results on an industrial system and section 9 provides the conclusion.

5.2. Features

Our goal is to apply program refactoring at the function level (method level). One of the proposed solutions is to consider each statement in a function as an independent entity, and each entity as having different attributes (data variables, loop counter variables, and control variables). Then, for each function, there will be an entity–attribute matrix. This matrix will form the input of the clustering technique. The clustering technique will output a set of clusters which will aid the designers to identify the low-cohesive functions and decompose them into several code fragments or to compose them into new functions.

5.3. Entities and Attributes

Our main concern is using software refactoring to increase cohesion. This procedure will increase the software understandability and thus the software quality. Entities are the items that need to be grouped. There are two types of statements: executable and non-executable statements. Non-executable statements, such as comments and declarations, have no real effect on the functionality provided by the function. Executable statements

include assignment statements, operational statements, and iteration statements. The restructuring approach will be applied only on the executable statements. We divide entities into control entities and non-control entities. A control entity refers to an entity that is either a predicate statement (such as the IF statement) or an iteration statement (such as the FOR statement). An attribute is a feature or property of an entity. An entity may have many attributes. Different properties of an entity can be described by different attributes. Attributes are used to calculate how closely two entities are related, based on the fact that entities are more similar if they share more common attributes.

In order to perform clustering on program statements, the attributes of the entity need to be identified. A statement consists of variables, constants, operators, keywords, brackets, function names and a semicolon. A statement is evaluated to see if it is related to a functional activity. Different variables and function names may be related to different functional activities, and therefore are used as attributes. Variables are divided into data variables and control variables. A data variable refers to the variable that is directly used in a statement. Data variables reveal the data dependence relationship of entities. A loop counter variable is used to count the number of times a loop is repeated. The loop body is treated as having the same relatedness to one or more functional activities. A control variable is artificially added to describe entities in a control block. It is a logical variable, used to describe the control dependence relationship between entities. Entities with the same control variable belong to the same control block in the source code (e.g., FOR block). An attribute can be measured with a quantitative scale or a qualitative scale. Based on our definition of attributes, each attribute is measured on a qualitative scale as a binary

representation. Thus, each attribute has two states: either presence or absence. If a data or a control attribute is absent, then the value of that attribute is '0'. If a control attribute is present, then the value of that attribute is '1', and if a data attribute is present, then the value of that attribute is '2'. Figure 5.1 shows an example of a sample function, and Table 5.1 shows the entity-attribute matrix of this sample code.

```
  void sum_and_prod(int n, int[] arr, int sum, int prod,
  float avg) {
1 sum = 0;
2 prod = 0;
3 for ( int i = 1 ; i < n ; i ++ )
      {
4     sum = sum + arr [ i ] ;
5     prod = prod * arr [ i ] ;
      }
6 avg = sum / n ;
7 System.out.println("The Prod="+prod);
8 Systme.out.println("The Avg="+avg);
}
```

Figure 5.1: Sample code

Entity/attribute	n	arr	Sum	prod	avg	i (for)
1	0	0	2	0	0	0
2	0	0	0	2	0	0
3	1	0	0	0	0	1
4	0	2	2	0	0	1
5	0	2	0	2	0	1
6	2	0	2	0	2	0
7	0	0	0	2	0	0
8	0	0	0	0	2	0

Table 5.1: Entity-Attribute matrix for the sample code in the Figure 5.1

The source code in Figure 5.1 is a simple function to calculate two values (sum, prod) based on the elements in an input array, and then to print them. The entity-attribute matrix for this sample of code is shown in Table 5.1. Every row in the table represents one of the

function lines. Hence, the 8 rows in the table represent the 8 lines of code in the functions. The columns in the table represent the attributes in the function. Thus, we have 6 columns that represent the 6 attributes in the function (n, arr, sum, prod, avg, for). The first 5 attributes (n, arr, sum, prod, avg) are data attributes. The first attribute (n) is the loop counter, and hence it is considered as data attribute. If a line of code contains any of these attributes, then a value of 2 will be put in the cell of the related row under the column that represents the attribute in the table. For example, the first row of the table represents the first line of code. This line of code contains the attribute *sum*. So, a value of 2 is put in the third column of this line. The last column in the table represent the variable *for*, which is a control variable. Thus, if any line of code satisfies this attribute (a line inside the FOR loop), then the value in the related row in the table and under the column which represents the attribute *for* will be 1. For example, line number 7 is inside the FOR loop. Hence, the value of row 7 under the last column is 1. Any two entities may have six different types of combinations or matches for each attribute, as Table 5.2 shows:

Combination	**Indication**
1–1 match	A control attribute is present in both entities.
2–2 match	A data attribute is present in both entities if neither of them is a control entity.
0–0 match	A data or control attribute is absent in both entities.

1–0 or 0–1 match (mismatch)	A control attribute is present in one entity and absent in the other.
2–0 or 0–2 match (mismatch)	A data attribute is present in one entity and absent in the other.
2–1 or 1–2 match	A data attribute is present in both entities. However, it is a control variable in one entity, and it is a non-control variable in the other (like the variable which contains the maximum number of iterations in a loop definition).

Table 5.2: Attributed matching combinations and their indications

5.4. Similarity measure

The more attributes two entities share, the closer they are related and the more similar they are. The resemblance coefficient between two entities is defined as follows [56]:

$$Coeff = (W_d * D_b + W_c * C_b) / (W_d * D_b + W_c * C_b + W_d * D_0 + W_c * C_0) \quad \ldots\ldots\ldots(1)$$

where

Coeff is the resemblance coefficient;

D_b is the number of 2–2 matches between two entities;

C_b is the number of 1–1 matches between two entities;

D_0 is the number of 2–0/0–2 matches between two entities;

C_0 is the number of 1–0/0–1 matches between two entities;

W_d is the weight of data attributes;

W_c is the weight of control attributes;

$W_d > W_c > 0$.

Here, the weight of an attribute represents its importance compared to the other attributes. The attributes of the same type have the same weight, and the weight of data attributes is heavier than that of control attributes. If no common attribute is shared by two entities, they are unrelated and Coeff = 0. If all attributes used to describe two entities are shared by them, $D_0 = 0$ and $C_0 = 0$, then they achieve the maximum similarity with Coeff=1. The value of the resemblance coefficient is between 0 and 1. However, in the real implementation we used dissimilarity. The dissimilarity measure is the complement of the resemblance coefficient.

$$Dis = 1 - Coeff$$

The dissimilarity measure will give us more flexibility in implementing the algorithm, and better insight on how the algorithm works.

5.5. Data Clustering

We used a numerical taxonomy or agglomerative hierarchical approaches. The hierarchical clustering algorithms SLINK, CLINK, WPGMA were described in chapter 3. The following subsections explain how we used SLINK, CLINK, and WPGMA. In

addition, we present our new algorithm Adaptive K- Nearest Neighbor (A-KNN) and we describe the main differences between A-KNN and traditional K- Nearest Neighbor (KNN).

5.5.1 SLINK, CLINK, WPGMA clustering

The most important aspect of hierarchical clustering is that the data is not partitioned into a particular cluster in a single step. Instead, a series of partitions takes place. It may run from a single cluster containing all objects to several clusters, each containing one or more object. All the SLINK, CLINK, WPGMA techniques have the same steps: Assigning each point to a cluster, and then merging the next two closest clusters until there is only one cluster. In SLINK, we calculate the distance between two groups as the distance between the closest pair of objects, taking one object from each group. The minimum value of these distances is considered as the distance between the two clusters. In CLINK, we take the distance between the most distant pair of objects, one from each group. The distance between every possible object pairs is computed, and the maximum value of these distances is taken. In average linkage clustering, the distance between two clusters is taken as the average of distances between all pairs of objects.

5.5.2. Adaptive K-Nearest Neighbor Clustering (A-KNN)

In this work, we adapted the K-NN clustering algorithm to suit the current problem. KNN was first introduced by Fix and Hodges [61]. In this algorithm, the space is partitioned

into regions by the locations and labels of the training samples. A point in the space is assigned to the class *c* if it is the most frequent class label among the *k* nearest training samples. KNN can be summarized in the following steps:

1. Store all input/output pairs in the training set
2. For each pattern in the test set
    a. Search for the *K* nearest patterns of the input patterns by using the Euclidean distance measure
    b. Compute the confidence for each class as C_i/K, where C_i is the number of patterns among the *K* nearest patterns belonging to class *i*
    c. The classification of the input pattern is the class with the highest confidence.

Figure 5.2 shows an example of KNN. The test sample (the circle) should be assigned either to the first class of "squares" or to the second class of "triangles". If $k = 3$ it is classified to the second class because there are 2 triangles and only 1 square inside the inner circle. If $k = 5$ it is classified to the first class (3 squares vs. 2 triangles inside the outer circle).

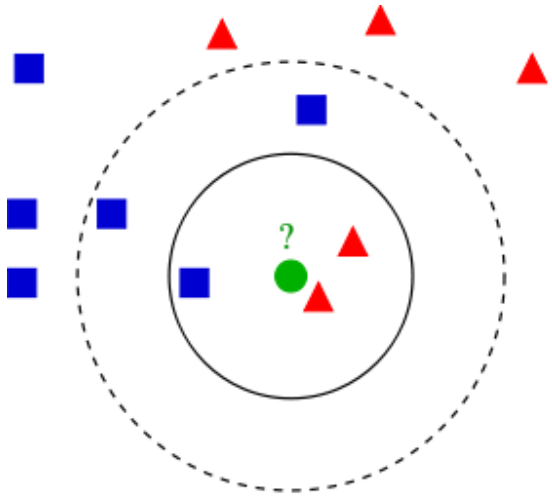


Figure 5.2: Example of KNN classification

A good *k* can be selected by various heuristic techniques, such as cross-validation. If K=1 then the class is predicted to be the class of the closest training sample. This is called the nearest neighbor algorithm (NN).

The implemented A-KNN clustering is as follows:

1) Obtain the data set.
2) Compute the resemblance coefficients for the data set.
3) Process the clustering assignment.
4) Repeat steps 2 and 3 until no more changes in clustering assignment are produced.

In the first step, the software code will be scanned to extract the intended attributes. These attributes are organized into a component–attribute data matrix. Components are the entities that need to be grouped. Attributes are the properties of the components. The

components and attributes can be many things in various areas, to which clustering analysis can be applied. In our implementation, the components are the program statements, and the attributes are the data and control variables.

In the second step, a resemblance coefficient for a given pair of components indicates the degree of similarity or dissimilarity between these two components, depending on the way in which the data is represented. For example, the data may be represented by means of a binary variable. If the value of the variable is '1', then the component has the property. However, if the data represents a misfit and the value of the variable is '1', then this will stand for one possible kind of misfit or dissimilarity. A resemblance coefficient can be qualitative or quantitative. A quantitative coefficient measures the literal distance between two components when they are viewed as points in a two-dimensional array formed by the input attributes. In our approach, the coefficient resemblance matrix will be calculated once. It will be used in all further iterations.

In step three, the algorithm starts by considering each entity as a cluster (i.e. each entity will be labeled with a unique identifier representing the cluster identity). In the second iteration, for k=3, the algorithm selects the three nearest neighbors to the entity that will be clustered, and then check their labels. If at least two out of the three clusters have the same label, the algorithm will label the current entity with the same label of those two entities. If the three entities have different labels, the algorithm will label the current entity with the same label of the closest entity.

In step 4, the algorithm repeats the second and third steps until no more changes occur in the clustering tree. This usually happens when the algorithm assigns all entities to the same cluster. Then the algorithm will output one cluster at the highest level of the hierarchy. The implemented A-KNN is shown in Figure 5.3.

Algorithm: A-KNN for K=3

Input: Entity-Attribute Matrix (n × m); n: number of entities, m: number of attributes

Output: Hierarchy of clusters.

1. Assign each entity to a single cluster and label each cluster. ($\mathcal{L}(C_i)$=*unique label, i belongs to {1,..,n}; where n is the number of entities*)
2. **While** the number of clusters is more than one **Do**
    a. **Calculate the Similarity Matrix (n × n)** (*Calculate the similarity between each cluster and all other clusters using the formula (1) and fill the n × n matrix).*
    b. **Find the most similar three pairs of clusters** $\{C_a, C_b\}$, $\{C_d, C_e\}$, $\{C_f, C_g\}$, where: *Coeff* (C_a, C_b) > *Coeff* (C_d, C_e) > *Coeff* (C_f, C_g)
    c. **If** $\mathcal{L}(C_a) = \mathcal{L}(C_d) = \mathcal{L}(C_f)$ (the same cluster) **Then**
        i. **If** $\mathcal{L}(C_e) = \mathcal{L}(C_g)$ **Then**
            1. $\mathcal{L}(C_a) = \mathcal{L}(C_e)$ *(merge clusters of* C_a *and* C_e *in one cluster)*
        ii. **Else**
            1. $\mathcal{L}(C_a) = \mathcal{L}(C_b)$ *(merge clusters of* C_a *and the* C_b *in one cluster* ***(normal clustering)****)*
    d. Else
        i. $\mathcal{L}(C_a) = \mathcal{L}(C_b)$ *(merge clusters of* C_a *and* C_b *in one cluster* ***(normal clustering)****)*
3. **End While**
4. **Return 'the hierarchy of clusters'**

Figure 5.3: Implemented A-KNN, for K=3

However, Step 2.a can be moved to be out of the **while** loop as the similarity matrix needs to be calculated once. Thus, the number of computations can be decreased significantly. The implemented A-KNN has an advantage over previous clustering techniques (SLINK, CLINK, WPGMA) because it reduces the amount of required computations. In A-KNN there is no need to calculate the distance between two clusters directly, whereas this consumed significant amounts of computation in the three previous algorithms. A-KNN depends mainly on the original similarity matrix between entities. Consequently, the amount of computation in each clustering step is less than SLINK, CLINK and WPGMA.

5.6. Refactoring approach using clustering techniques

Figure 5.4 shows the approach for program refactoring by using clustering techniques. The current study uses programs developed with java; however, the technique can be applied to other languages as well. The approach has three phases as shown in Figure 5.4.

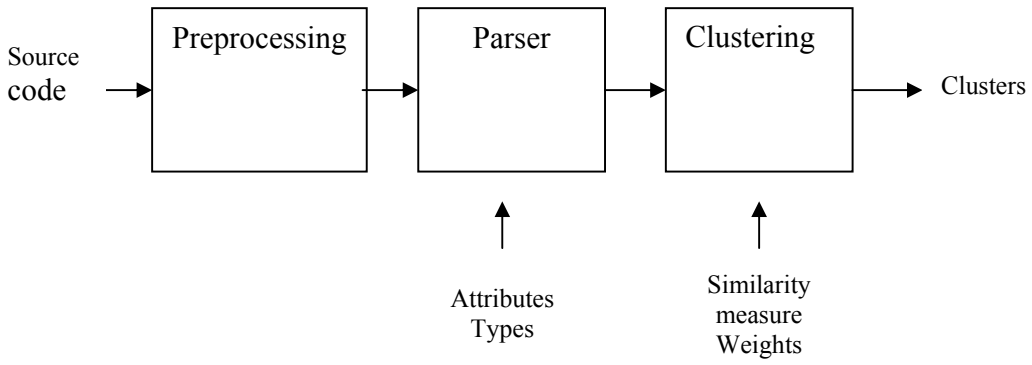


Figure 5.4: Refactoring Process

Phase I is data processing. Our data is the source code of the software. In fact, each software developer has his own way of writing code. A code can be written in many ways, and all of them will be acceptable for the programming language compiler. Having a code written in different ways may complicate the parser which we will use in the second step. To simplify the work of the parser, we add a preprocessor at the first phase to rewrite the code more simply. The parser extracts the data from the source code by reading the source code as tokens. Consequently, if the programmer did not leave spaces between the tokens the parser may consider two successive tokens as one token.

```
     void sum_and_prod(int n,int[] arr,int sum,int
     prod,float avg) {
1    sum = 0;prod = 0;
2    for(int i=1;i<n;i ++)
     {
3    sum=sum+arr[i];prod=prod*arr [i];
     }
4    avg=sum / n ;
     }
```

Figure 5.5: Sample code

For example, if we have the code in Figure 5.5, the data extraction will be complicated because most of the variables, commas, operations and other words are attached to each other. This code is syntactically correct. In other words, the compiler will not give any error message during the compilation process. However, the parser needs more operations to extract the required data. For example, in the header of the function, the parser will consider the two words “sum,int”, as one token, and hence the parser needs to manipulate the previous string character-by-character to extract the data. Another example is the line

3 which contains two lines of code. However, the developer put no space after the semicolon. To overcome this problem, preprocessing will be used in the first phase to rewrite the previous code in a simplified way as shown in Figure 5.6.

```
void sum_and_prod(int n, int[] arr, int sum, int prod,
float avg) {
sum = 0;
prod = 0;
for ( int i = 1 ; i < n ; i ++ )
{
    sum = sum + arr [ i ] ;
    prod = prod * arr [ i ] ;
}
  avg = sum / n ;
}
```

Figure 5.6: Sample code after preprocessing

Phase II is data collection and processing. In this phase, the parser tool parses the source code automatically, and it generates the entity–attribute matrix. Entities are the items or lines to be clustered. Each entity has one or more attributes. Entities are grouped by the attributes they share. Thus, the more attributes two entities have in common, the more closely they are related. In order to apply the clustering technique to programs, we first need to define entities and attributes specifically relevant to the functions in this phase.

Phase III is clustering. In this phase we provide the clustering techniques with the similarity measures' weights. After entities and their attributes are defined, the resemblance coefficients are calculated to measure the similarity between every two entities. After the resemblance coefficients are defined, clusters can be constructed by

using a clustering algorithm. Four hierarchical agglomerative algorithms SLINK, CLINK and WPGMA, A-KNN will be investigated.

The clustering tree provides heuristic advice (aid) on how to restructure a function. Software designers must participate in making the final decision based on their experience, insights, and the restructuring objective.

5.7. Experiments on Code Extracted from Published Papers

We tested the above techniques on code taken from published work in the literature. The code which we used was mainly on published papers related to software design and the cohesiveness of the code. The code was written in Fortran, but we wrote it again using Java. This enables us to compare the results of A-KNN with the published clustering techniques reported in the literature.

In this section we will present our results. First, we will present the results of the comparative study among the three agglomerative clustering algorithms. Then, we will present the results of the second comparative study between those three algorithms and our introduced algorithm (A-KNN).

5.7.1. Experimental Results on SLINK, CLINK and WPGMA

Three hierarchical agglomerative algorithms: SLINK, CLINK and WPGMA, are chosen and a comparative study on them is conducted. These experiments are briefly described as follows:

5.7.1.1. Weight 2:1

The weight of data and control variables needs to be investigated to choose the best weight for the data and control variables. However, Dhama [57] presented a heuristic estimate to consider each data attribute equivalent to two control variables. In other words, the data variable has double the weight of the control variable. A further contribution in this domain was presented by Schwanke [50] to estimate the feature significance using Shannon information content. Shannon information content gives less weight to frequently used identifiers than to rarely-used identifiers. In addition, Lung et al. [56] considered the weight of attributes as positive integers, and they decided the weight through extensive experiments. They gave the data variable 8 units and the control variable 3 units. In the first experiment we tried the ratio 2:1 as the weight for data: control variables. Figure 7.5 shows the source code we used in our experiments.

```
     void sum_and_prod(int n, int[] arr, int sum, int
     prod, float avg) {
1    sum = 0;
2    prod = 0;
3    for ( int i = 1 ; i < n ; i ++ )
     {
4    sum = sum + arr [ i ] ;
5    prod = prod * arr [ i ] ;
     }
6    avg = sum / n ;
     }
```

Figure 5.7: Sample code of reference [56]

The desirable output from the clustering algorithm is putting the lines 1, 4, 6 in the same cluster as they share the '*sum*' variables. Lines 2 and 5 share the '*prod*' variable, and these lines are expected to be in the same cluster. Thus, after 3 steps of clustering, it is expected to have two clusters {{1, 4}, 6} and {2, 5}. Line 3 shares the 'for' control variable with the second cluster {2, 5} and it shares the 'for' control variable with the line 4 which is already in the cluster {{1, 4}, 6}. It also shares the 'n' variable with line 6 which is also in the cluster {{1, 4}, 6}. Consequently, the clustering technique will add it to one of the two clusters based on the weights given to data and control variables. Thus, the clustering hierarchy tree aids the software designer in taking the decision to split this function into other functions. The first contains Lines 1, 4 and 6 and second contains lines 2 and 5. Moreover, the clustering hierarchy tree will show that line 3 is similar to the two clusters. Table 5.3 shows the similarity matrix values for the sample code of Figure 5.7.

	1	2	3	4	5	6
1	1.0	0.0	0.0	0.4	0.0	0.33
2	0.0	1.0	0.0	0.0	0.4	0.0
3	0.0	0.0	1.0	0.17	0.17	0.0
4	0.4	0.0	0.17	1.0	0.43	0.22
5	0.0	0.4	0.17	0.43	1.0	0.0
6	0.33	0.0	0.0	0.22	0.0	1.0

Table 5.3: Similarity Matrix of the code in the Figure 5.7 with 2:1 weight ration

For example, the entities 4, 5 have: 1: 2-2 match, 2: 2-0/0-2 matches, 1: 1-1 matches, 0: 1-0/1-0 match. The similarity between the two lines can be calculated by using the following formula:

$$Coff_{(4,5)}= (1*2 + 1*1)/ ((1*2 + 1*1)+(2*2+ 0*1))=0.43$$

Thus the similarity between the clusters 4 and 5 is 0.43.

$$Diss\text{-}Coff_{(4,5)}=1\text{-}Coff_{(4,5)}=1 - 0.43=0.57$$

Table 5.4 shows the values of dissimilarity matrix for the sample code in the Figure 5.7.

	1	2	3	4	5	6
1	0.0	1.0	1.0	0.6	1.0	0.67
2	1.0	0.0	1.0	1.0	0.6	1.0
3	1.0	0.0	0.83	0.83	1.0	1.0
4	0.6	1.0	0.83	0.0	0.57	0.78
5	1.0	0.6	0.83	0.57	0.0	1.0
6	0.67	1.0	1.0	0.78	1.0	0.0

Table 5.4: Dissimilarity Matrix of the code in the Figure 5.7 with 2:1 weight ratio

Figure 5.8, Figure 5.9 and Figure 5.10 show the output of the experiments for algorithms SLINK, CLINK and WPGMA, respectively. In the first step, the algorithm considers each entity as a separate cluster. In the next step, the algorithm merges the clusters {5} and {4} because they have the minimum dissimilarity value in the dissimilarity matrix (0.57). In the next step, since the dissimilarity between the new cluster {4,5} and cluster {1} is the next minimum dissimilarity in the matrix (0.6), the algorithm added cluster {1} to the

new cluster {4,5} which becomes cluster {{4,5},1}. In the next step, the dissimilarity between cluster {2} and the cluster {{4,5},1} is (0.6) which is the next minimum value in the dissimilarity matrix. Thus, the algorithm added the cluster {2} to cluster {{4,5},1} which becomes cluster {{{4,5},1}, 2}. The algorithm continues until all clusters are merged into one cluster. However, the figures show that none of the algorithms gave the expected output.

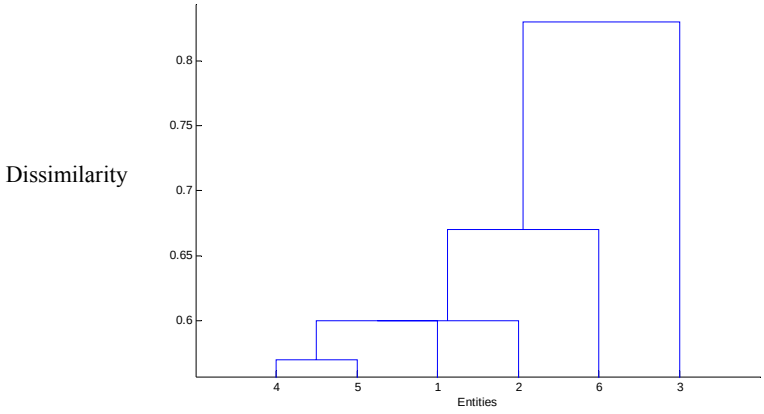


Figure 5.8: Clustering tree of SLINK with 2:1 weight ratio

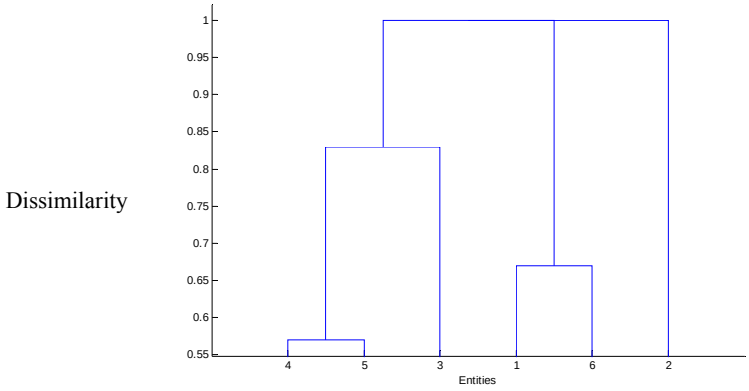


Figure 5.9: Clustering tree of CLINK with 2:1 weight ratio

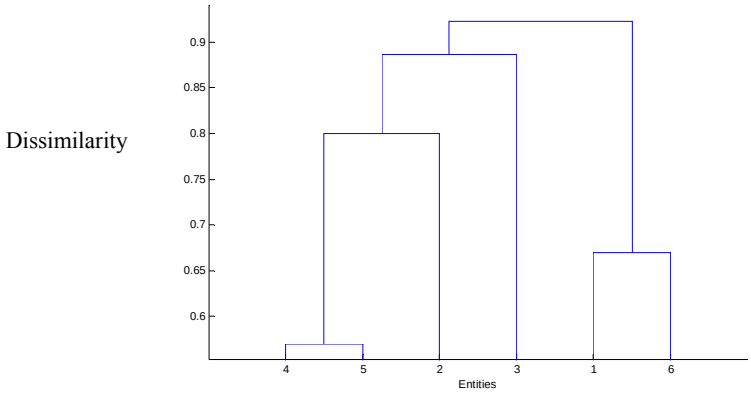


Figure 5.10: Clustering tree of WPGMA with 2:1 weight ratio

5.7.1.2. Weight 3:1

In the second experiment, we tried the ratio 3:1 as the weight for data: control variables. We used the same source code as in Figure 5.7. Table 5.5 and Table 5.6 show the similarity and dissimilarity matrix respectively. Figure 5.11, Figure 5.12 and Figure 5.13 show the output of the experiments. The weight 3:1 proved to be better than weight 2:1. However, the output of the clustering phase is not expected.

	1	2	3	4	5	6
1	1.0	0.0	0.0	0.43	0.0	0.33
2	0.0	1.0	0.0	0.0	0.43	0.0
3	0.0	0.0	1.0	0.12	0.12	0.0
4	0.43	0.0	0.12	1.0	0.4	0.23
5	0.0	0.43	0.12	0.4	1.0	0.0
6	0.33	0.0	0.0	0.23	0.0	1.0

Table 5.5: Similarity Matrix of the code in the Figure 5.7 with 3:1 weight ratio

	1	2	3	4	5	6
1	0.0	1.0	1.0	0.57	1.0	0.67
2	1.0	0.0	1.0	1.0	0.57	1.0
3	1.0	1.0	0.0	0.88	0.88	1.0
4	0.57	1.0	0.88	0.0	0.6	0.77
5	1.0	0.57	0.88	0.6	0.0	1.0
6	0.67	1.0	1.0	0.77	1.0	0.0

Table 5.6: Dissimilarity Matrix of the code in the Figure 5.7 with 3:1 weight ratio

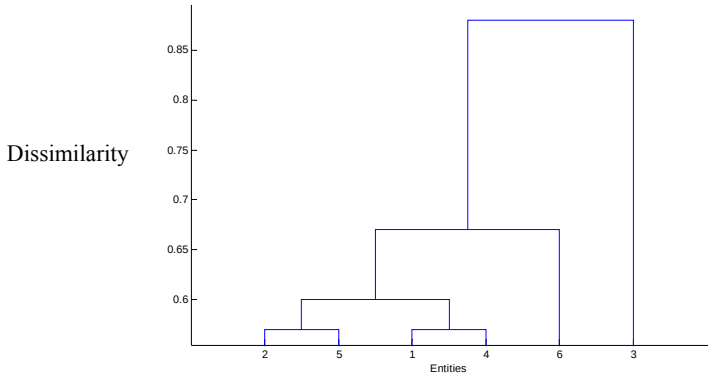


Figure 5.11: Clustering tree of SLINK with 3:1 weight ratio

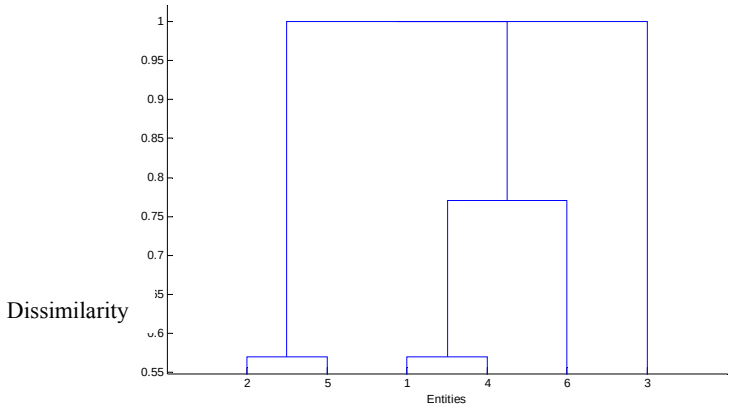


Figure 5.12: Clustering tree of CLINK with 3:1 weight ratio

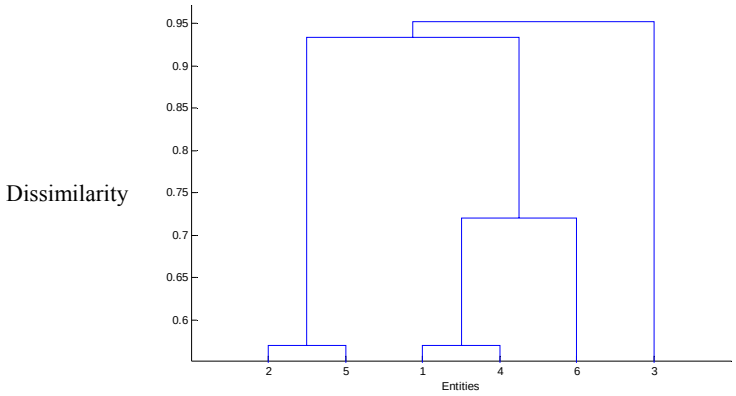


Figure 5.13: Clustering tree of WPGMA with 3:1 weight ratio

5.7.1.3. Weight 8:3

In the third experiment we tried the ratio 8:3 as the weight for data: control variables. We used the same source code as in Figure 5.7. Table 5.7 and Table 5.8 show the similarity and dissimilarity matrix, respectively. Figure 5.14, Figure 5.15 and Figure 5.16 show the output of the experiment. Figure 5.16 shows that the WPGMA algorithm gave us the desirable output since it merged lines 1, 4 and 6 in one cluster and 4, 5 in another cluster. According to the previous experimental results, WPGMA was the best algorithm and 8:3 was the best weight.

	1	2	3	4	5	6
1	1.0	0.0	0.0	0.42	0.0	0.33
2	0.0	1.0	0.0	0.0	0.42	0.0
3	0.0	0.0	1.0	0.14	0.14	0.0
4	0.42	0.0	0.14	1.0	0.41	0.23
5	0.0	0.42	0.14	0.41	1.0	0.0
6	0.33	0.0	0.0	0.23	0.0	1.0

Table 5.7: Similarity Matrix of the code in the Figure 5.7 with 8:3 weight ratio

	1	2	3	4	5	6
1	0.0	1.0	1.0	0.58	1.0	0.67
2	1.0	0.0	1.0	1.0	0.58	1.0
3	1.0	1.0	0.0	0.86	0.86	1.0
4	0.58	1.0	0.86	0.0	0.59	0.77
5	1.0	0.58	0.86	0.59	0.0	1.0
6	0.67	1.0	1.0	0.77	1.0	0.0

Table 5.8: Dissimilarity Matrix of the code in the Figure 5.7 with 8:3 weight ratio

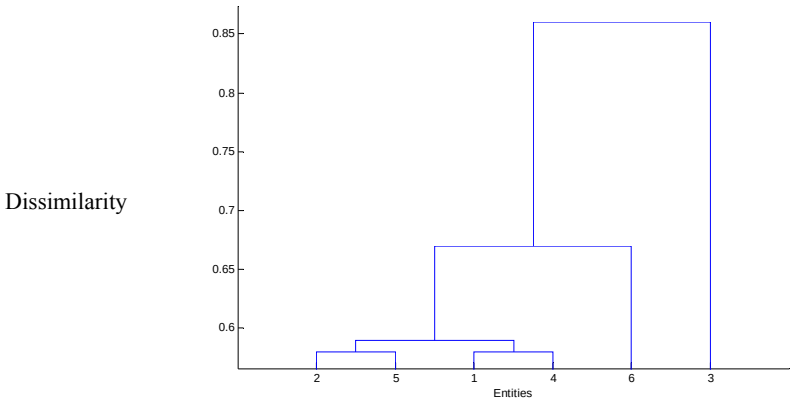


Figure 5.14: Clustering tree of SLINK with 8:3 weight ratio

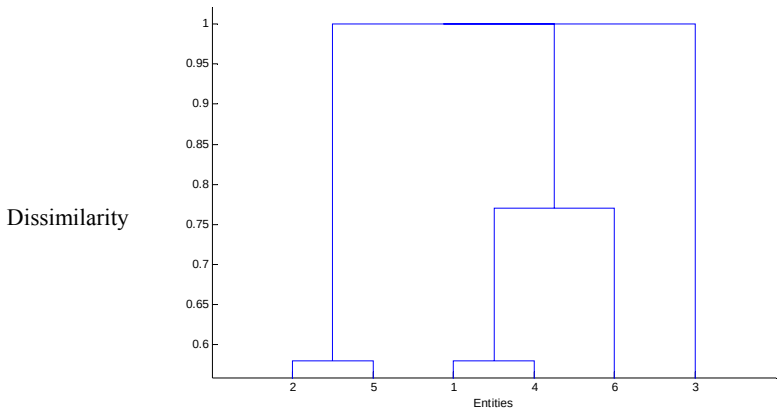


Figure 5.15: Clustering tree of CLINK with 8:3 weight ratio

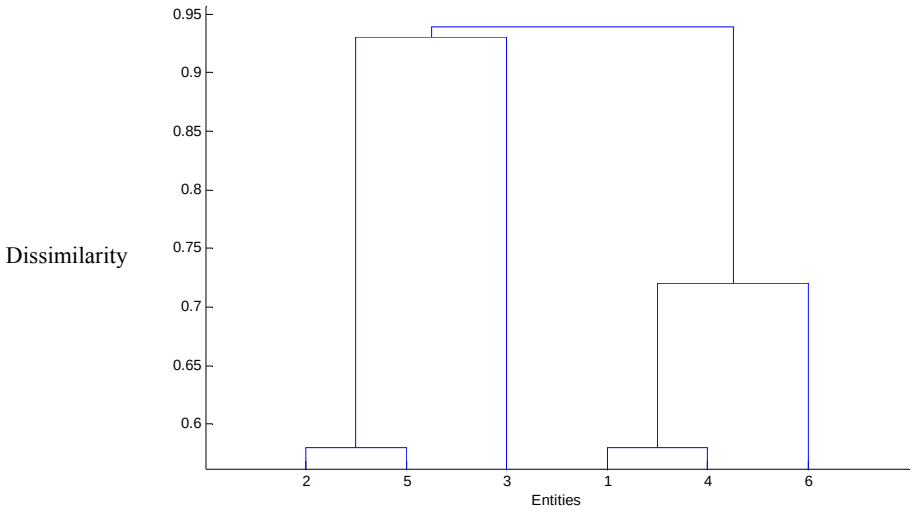


Figure 5.16: Clustering tree of WPGMA with 8:3 weight ratio

5.7.2. Experimental Results on SLINK, CLINK, WPGMA and A-KNN

In this experiment we compared the performance of the three previous algorithms with our implemented (A-KNN) algorithm for K=3. However, if the size of the code is large, 5-NN may be needed. The code which we selected for the comparison is shown in Figure 5.17. The function in Figure 5.17 which is extracted from reference [25] reads the amount of sales per day, for a given number of days, and it computes (a) the total_sales for the period, (b) the commission to be paid (i.e. 10% of the sales, with an added bonus of $50 if the sales for a particular day exceed $1,000) and (c) the profit for the whole period (given the cost at the beginning, as a percentage of sales). Although the function is small, it performs several activities which make it non-cohesive. Thus, this part of the code can be described as ill-structured code.

```
     void Sale_Pay_Profit ( int days, float cost,
     int [] sale, double pay, double profit, boolean
     process) {
1    while ( i < days )
     {
2    i = i + 1 ;
3    sale [ i ] = my_read();
     }
4    if ( process == true )
     {
5    int total_sale = 0;
6    int total_pay = 0;
7    for ( i = 1 ; i < days ; i ++)
     {
8    total_sale = total_sale + sale [ i ];
9    total_pay = total_pay + 0.1 * sale [ i ];
10   if ( sale [ i ] > 1000 )
11   total_pay = total_pay + 50;
     }
12   pay = total_pay / days + 100;
13   profit = 0.9 * total_sale - cost ;
     }
     }
```

Figure 5.17: Sample code extracted from reference [25]

Figure 5.8 contains a program equivalent to that of Figure 5.17. Though it is larger in size, as measured by the number of lines, this program is cohesive [25]. Thus, instead of writing one function that performs several activities, it has several small functions, each performing a single activity. In software design, it is accepted that several functions with a small number of lines of code, where each function performs a small task, will be easier to understand and easier to maintain [25]. The function in Figure 5.17 is an example of code with interleaved computations [59]. In other words, this function uses the same input to compute the different outputs. The difficulty in decomposing large code fragments lies not so much in creating small functions, but in creating small functions that are

meaningful [25]. Considering the following story, we can understand the difficulties associated with such code and how it will affect future maintenance on this code:

> "In the late 1960s most data processing managers began to recognize the benefits of modularity. Unfortunately many existing programs were monolithic, e.g., 20,000 lines of undocumented FORTRAN program with one subroutine of 2500 lines of code. To bring his environment to the state of the art, a manager asked his staff to modularize such a program that underwent maintenance continuously. This was to be done "in your spare time." Under the gun, one staff member asked (innocently) the proper length for a module. "Seventy-five lines of code," came the reply. She then obtained a red pen and a ruler, measured the linear distance taken by 75 lines of source code, and drew a red line on the source listing, then another and another. Each red line indicated a module boundary [60, page 334]."

While this approach appears humorous, it highlights the problem in decomposing code in which a set of statements could be extracted as independent functions or procedures [25]. Although object-oriented metrics, such LCOM, are able to explain the degree of cohesion at the class level, it will not be a suitable measure to show the advantages of the code in Figure 5.18 over the code in Figure 5.17. This is because applying such a measure on the two source codes will give us the cohesion at the class level, not the function level. Consequently, a class containing one method will be more cohesive than a class containing many functions, regardless of the cohesiveness of those functions. For

example, if we measure the LCOM for class that contains the function in Figure 5.17 it will be 0, but for the class in the Figure 5.18 it will be 15. Thus, the measures of the object-oriented metric could give us misleading results in such cases.

```
package refactoring;

public class Sale {
double pay;
double profit;
int[] sale;

  public Sale() {
  }

  void read_Input(int days)  {
  int i;
  i=0;
  while (i < days){
  i = i + 1;
  this.sale[i]=my_read(i);
  }
  }

  double compute_Pay(int days, int[] sale)  {
      double total_pay;
      int j;
      total_pay = 0;
      for (j = 1; j < days; j++) {
        total_pay = total_pay + 0.1 * sale[j];
        if (sale[j] > 1000)
          total_pay = total_pay + 50;
      }
      return total_pay;
  }

  double compute_Sale(int days, int[] sale) {
        double total_sale;
        int j;
        total_sale = 0;
        for (j = 1; j < days; j++) {
      total_sale = total_sale + sale[j];
    }
    return total_sale;
    }

  double compute_Avg_Pay(int days, int[] sale) {
  double total_pay;
  double pay;
  total_pay = compute_Pay(days, sale);
  pay = total_pay / days + 100;
  return pay;
  }

  double compute_Profit (double cost, int[] sale, int days)  {
  double total_sale, profit;
  total_sale = compute_Sale(days, sale);
  profit = 0.9 * total_sale - cost;
  return profit;
  }

  void sale_Pay_Profit (int days, double cost, int[] sale,double pay, double
profit, boolean process)   {
    read_Input(days);
    if (process ==true)
    {
    this.pay = compute_Avg_Pay(days, sale);
    this.profit = compute_Profit(cost, sale, days);
    }
    }
}
```

Figure 5.18: Results expected from refactoring program in Figure 17

5.7.2.1. Weight 8:3

The ratio 8:3 was previously proved to be the best weight for the previous example. We will use the same weight in this experiment. Table 5.9 and Table 5.10 show the similarity and dissimilarity matrix, respectively. Figure 5.19, Figure 5.20 and Figure 5.21 show the output of the experiment.

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	1.0	0.5	0.21	0.0	0.0	0.0	0.25	0.0	0.0	0.0	0.0	0.0	0.0
2	0.5	1.0	0.27	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.21	0.27	1.0	0.0	0.0	0.0	0.0	0.47	0.47	0.4	0.0	0.0	0.0
4	0.0	0.0	0.0	1.0	0.27	0.27	0.18	0.14	0.14	0.12	0.18	0.11	0.11
5	0.0	0.0	0.0	0.27	1.0	1.0	0.33	0.21	0.21	0.18	0.33	0.16	0.16
6	0.0	0.0	0.0	0.27	1.0	1.0	0.33	0.21	0.21	0.18	0.33	0.16	0.16
7	0.25	0.0	0.0	0.18	0.33	0.33	1.0	0.35	0.35	0.3	0.5	0.21	0.12
8	0.0	0.0	0.47	0.14	0.21	0.21	0.35	1.0	1.0	0.82	0.35	0.1	0.1
10	0.0	0.0	0.47	0.14	0.21	0.21	0.35	1.0	1.0	0.82	0.35	0.1	0.1
11	0.0	0.0	0.4	0.12	0.18	0.18	0.3	0.82	0.82	1.0	0.53	0.09	0.09
12	0.0	0.0	0.0	0.18	0.33	0.33	0.5	0.35	0.35	0.53	1.0	0.12	0.12
13	0.0	0.0	0.0	0.11	0.16	0.16	0.21	0.1	0.1	0.09	0.12	1.0	0.09
14	0.0	0.0	0.0	0.11	0.16	0.16	0.12	0.1	0.1	0.09	0.12	0.09	1.0

Table 5.9: Similarity Matrix for the sample code in figure 17

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	0.0	0.5	0.79	1.0	1.0	1.0	0.75	1.0	1.0	1.0	1.0	1.0	1.0
2	0.5	0.0	0.73	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
3	0.79	0.73	0.0	1.0	1.0	1.0	1.0	0.53	0.53	0.6	1.0	1.0	1.0
4	1.0	1.0	1.0	0.0	0.73	0.73	0.82	0.86	0.86	0.88	0.82	0.89	0.89
5	1.0	1.0	1.0	0.73	0.0	0.0	0.67	0.79	0.79	0.82	0.67	0.84	0.84
6	1.0	1.0	1.0	0.73	0.0	0.0	0.67	0.79	0.79	0.82	0.67	0.84	0.84
7	0.75	1.0	1.0	0.82	0.67	0.67	0.0	0.65	0.65	0.7	0.5	0.79	0.88
8	1.0	1.0	0.53	0.86	0.79	0.79	0.65	0.0	0.0	0.18	0.65	0.9	0.9
9	1.0	1.0	0.53	0.86	0.79	0.79	0.65	0.0	0.0	0.18	0.65	0.9	0.9
10	1.0	1.0	0.6	0.88	0.82	0.82	0.7	0.18	0.18	0.0	0.47	0.91	0.91
11	1.0	1.0	1.0	0.82	0.67	0.67	0.5	0.65	0.65	0.47	0.0	0.88	0.88
12	1.0	1.0	1.0	0.89	0.84	0.84	0.79	0.9	0.9	0.91	0.88	0.0	0.91
13	1.0	1.0	1.0	0.89	0.84	0.84	0.88	0.9	0.9	0.91	0.88	0.91	0.0

Table 5.10: Dissimilarity Matrix for the sample code in figure 17

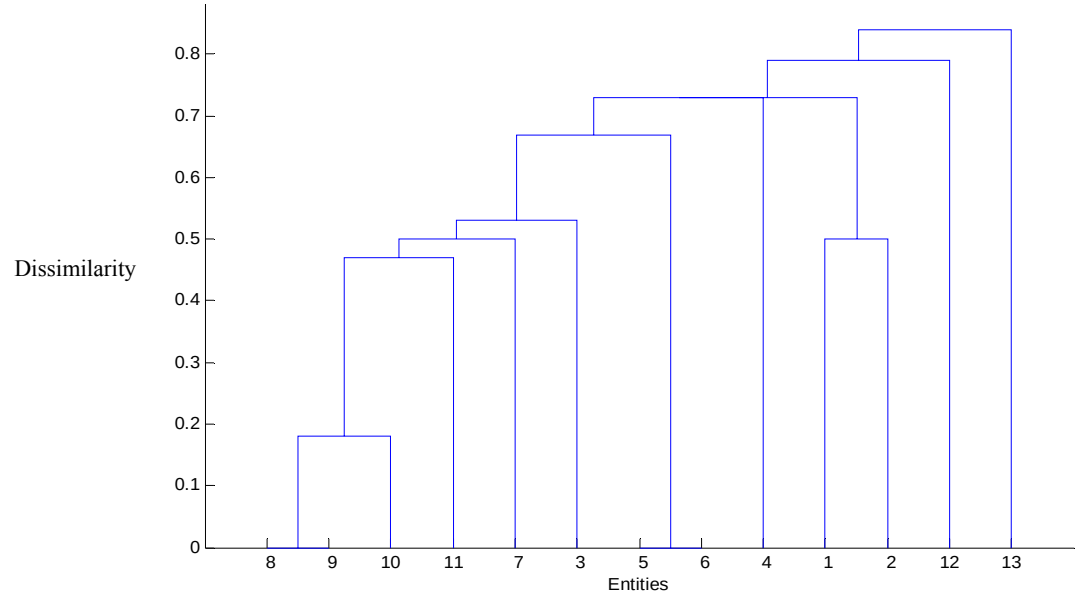


Figure 5.19: Clustering tree of SLINK for the sample code in figure 17 with 8:3 weight ratio

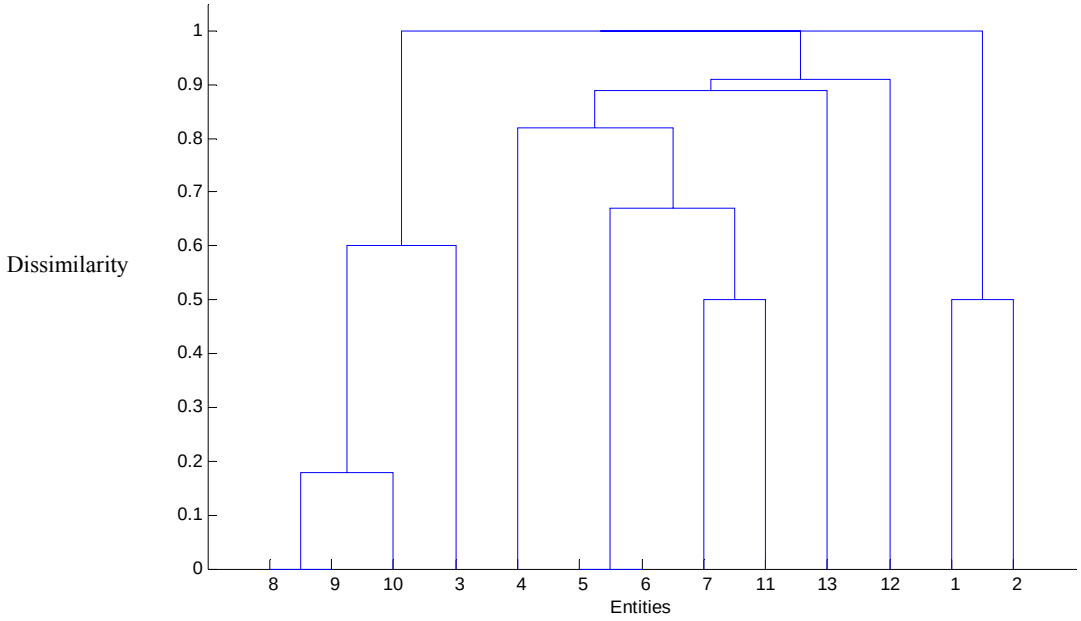


Figure 5.20: Clustering tree of CLINK for the sample code in figure 17 with 8:3 weight ratio

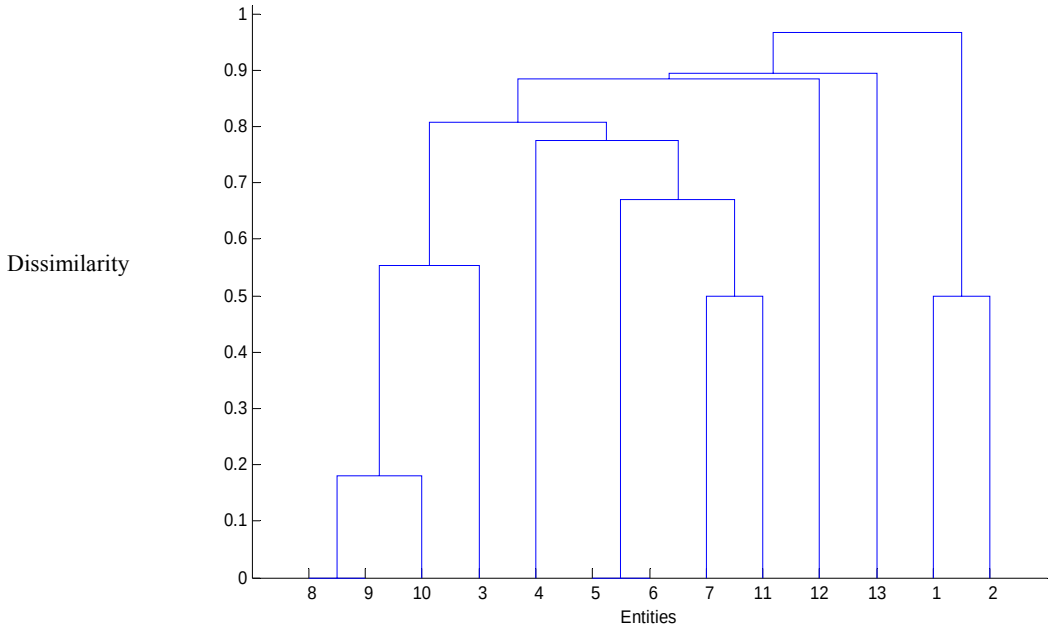


Figure 5.21: Clustering tree of WPGMA for the sample code in figure 17 with 8:3 weight ratio

We will apply A-KNN in successive steps. In each step it will give a new combination of the entities which we need to cluster. Table 5.11 shows the Adaptive 3-Nearest Neighbor technique output in each step, and Figure 5.22 shows the clustering tree of A-KNN. The algorithm starts by assigning each line of code to a single cluster in step 0. Thus, there will be 13 clusters at the end of step 0. In all other steps, the algorithm will choose the two most similar clusters, and merge them into a single cluster. For example, in step 1 the algorithm chose clusters 5 and 6 because they are the most similar (Dissimilarity =0) and it merged them into a new cluster. The algorithm gives a unique ID to the new cluster. Since there are already 13 clusters, the new cluster ID will be 14. In step 2, the algorithm merges clusters 8 and 9 into cluster 15, and so on until all clusters are merged into one cluster (with the ID 25).

Step	First clusters	Second Cluster	Dissimilarity	New Cluster
1	5	6	0	14
2	8	9	0	15
3	15	10	0.18	16
4	16	11	0.47	17
5	1	2	0.5	18
6	17	7	0.5	19
7	19	3	0.53	20
8	14	20	0.67	21
9	18	21	0.73	22
10	22	4	0.73	23
11	23	12	0.79	24
12	24	13	0.84	25

Table 5.11: Output of A-KNN for the sample code in figure 14 with K=3 and 8:3 weight ratio in 12 steps

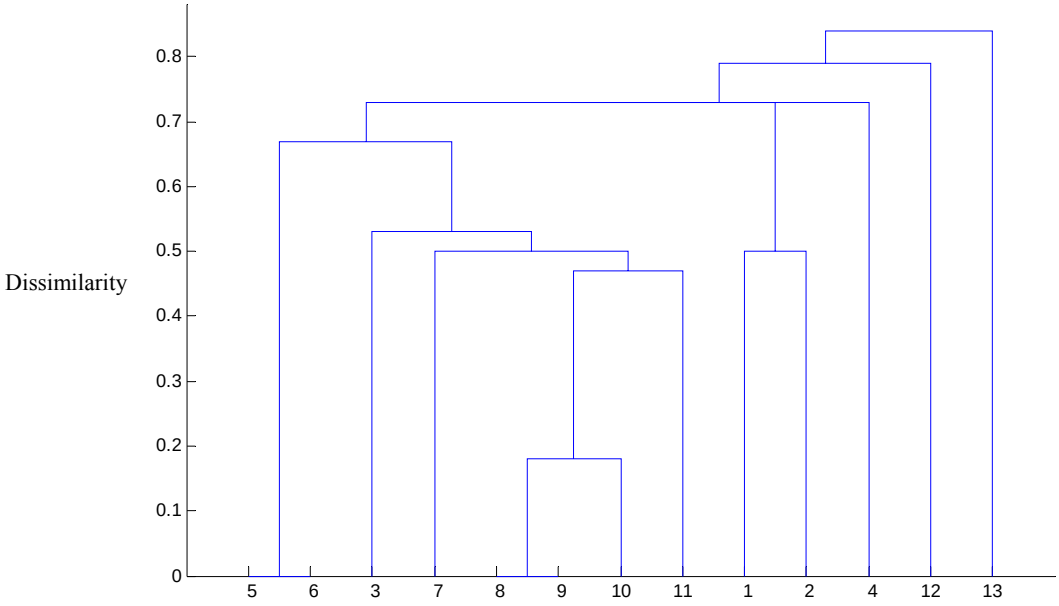


Figure 5.22: Clustering tree of A-KNN for the sample code in figure 17 for K=3 with 8:3 weight ratio

The previous results show that A-KNN performance is competitive with all algorithms. Moreover, in some steps it has a better performance than WPGMA which was proved previously to be the best algorithm of the three. Another benefit is that A-KNN requires less computation time.

5.8. Experimental Results on Industrial System

We applied our work on different source codes, by using the Java Development Kit 1.4 (JDK 1.4). We selected some classes of JDK and applied the clustering techniques. However, JDK proved to be unsuitable experimental unit for two reasons. The first reason is the nature of the code itself. The code is intended to provide an environment to run java programs. Thus, the functionality of code is totally different from the functionality of industrial systems. The second reason is that Sun Microsystems, being one of the largest companies in the world, probably uses a high level of standards in writing code with fully qualified developers. Thus, the probability of having ill-structured or low cohesive code is low.

Then, we applied the A-KNN technique on another real system. The system which we selected for this purpose is PDF Spilt and Merge (PDFSM). PDFSM is an easy-to-use tool to merge and split pdf documents. A console and GUI versions are available. The GUI is written in Java Swing, and it provides functions to select files and set options. It is made over the iText library. This system is available at [58]. The system consists of 160 java classes. We selected one of the largest functions in this system. Figure 5.23 shows the function that we applied the approach to.

```
public void execute(AbstractParsedCommand parsedCommand) throws ConsoleException {
0if((parsedCommand != null) && (parsedCommand instanceof MixParsedCommand)){
1       MixParsedCommand inputCommand = (MixParsedCommand) parsedCommand;
2       setWorkIndeterminate();
3       Document pdfDocument = null;
4       PdfCopy  pdfWriter = null;
```

```
5       PdfReader pdfReader1;
6       PdfReader pdfReader2;
7       int[] limits1 = {1,1};
8        int[] limits2 = {1,1};
9       try{
10      File tmpFile = FileUtility.generateTmpFile(inputCommand.getOutputFile());
11      pdfReader1 = new PdfReader(new
        RandomAccessFileOrArray(inputCommand.getFirstInputFile().getFile().getAbsolutePath(
        )), inputCommand.getFirstInputFile().getPasswordBytes());
12      pdfReader1.consolidateNamedDestinations();
13      limits1[1] = pdfReader1.getNumberOfPages();
14      pdfReader2 = new PdfReader(new RandomAccessFileOrArray
        (inputCommand.getSecondInputFile().getFile().getAbsolutePath()),inputCommand.getSec
        ondInputFile().getPasswordBytes());
15      pdfReader2.consolidateNamedDestinations();
16      limits2[1] = pdfReader2.getNumberOfPages();
17      pdfDocument = new Document(pdfReader1.getPageSizeWithRotation(1));
18      log.debug("Creating a new document.");
19      pdfWriter = new PdfCopy(pdfDocument, new FileOutputStream(tmpFile));
20      if(inputCommand.getOutputPdfVersion() != null){
21              pdfWriter.setPdfVersion(inputCommand.getOutputPdfVersion().charValue());
        }
22      if(inputCommand.isCompress()){
23                      pdfWriter.setFullCompression();
        }
24      pdfDocument.addCreator(ConsoleServicesFacade.CREATOR);
25      pdfDocument.open();
26      PdfImportedPage page;
27      boolean finished1 = false;
28      boolean finished2 = false;
29      int current1 = (inputCommand.isReverseFirst())? limits1[1] :limits1[0];
30      int current2 = (inputCommand.isReverseSecond())? limits2[1] :limits2[0];
31      while(!finished1 || !finished2){
32              if(!finished1){
33                      if(current1>=limits1[0] && current1<=limits1[1]){
34                              page = pdfWriter.getImportedPage(pdfReader1, current1);
35                              pdfWriter.addPage(page);
36      current1 = (inputCommand.isReverseFirst())? (current1-1):(current1+1);
37                      }else{
38                              log.info("First file processed.");
39                              pdfReader1.close();
40                              finished1 = true;
                        }
                }
41              if(!finished2){
42                      if(current2>=limits2[0] && current2<=limits2[1] && !finished2){
43                              page = pdfWriter.getImportedPage(pdfReader2, current2);
44                              pdfWriter.addPage(page);
45      current2 = (inputCommand.isReverseSecond())? (current2-1) :(current2+1);
46                      }else{
47                      log.info("Second file processed.");
48                      pdfReader2.close();
49                      finished2 = true;
                        }
                }
        }//while
50      pdfWriter.freeReader(pdfReader1);
51      pdfWriter.freeReader(pdfReader2);
52      pdfDocument.close();
53      if(FileUtility.renameTemporaryFile(tmpFile,inputCommand.getOutputFile(),
        inputCommand.isOverwrite())){
54       log.debug("File "+inputCommand.getOutputFile().getCanonicalPath()+" created.");
                }
55      log.info("Alternate mix completed.");
56      }catch(Exception e){
57              throw new MixException(e);
58      }finally{
59              setWorkCompleted();
```

```
        }
60      }else{
61              throw new ConsoleException(ConsoleException.ERR_BAD_COMMAND);
        }
        }
```

Figure 5.23: Source code of the function *execute* of reference [58]

Table 5.12 shows the entity-attribute matrix for the function *Execute* of the class *AlternateMixCmdExecutor* class. The columns represent the data and control attributes, and the rows represent the lines of code. If the line satisfies the attribute the value will be none zero (0 or 1). If the line does not satisfy the attribute, the value will be zero.

Line of code	parsedCommand	If 0	Try 9	If 20	If 22	While 31	If 32	If 33	Else 37	If 41	If 42	If 46	If 53	catch 56	Finaly 58	Else 60
0	2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
15	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
16	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
17	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
18	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
19	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
20	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
21	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
22	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
23	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0
24	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
25	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
26	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0

27	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
28	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
29	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
30	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
31	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0
32	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0
33	0	1	1	0	0	1	1	1	0	0	0	0	0	0	0	0
34	0	1	1	0	0	1	1	1	0	0	0	0	0	0	0	0
35	0	1	1	0	0	1	1	1	0	0	0	0	0	0	0	0
36	0	1	1	0	0	1	1	1	0	0	0	0	0	0	0	0
37	0	1	1	0	0	1	1	1	0	1	0	0	0	0	0	0
38	0	1	1	0	0	1	1	1	0	1	0	0	0	0	0	0
39	0	1	1	0	0	1	1	1	0	1	0	0	0	0	0	0
40	0	1	1	0	0	1	1	1	0	1	0	0	0	0	0	0
41	0	1	1	0	0	1	0	0	0	1	0	0	0	0	0	0
42	0	1	1	0	0	1	0	0	0	1	1	0	0	0	0	0
43	0	1	1	0	0	1	0	0	0	1	1	0	0	0	0	0
44	0	1	1	0	0	1	0	0	0	1	1	0	0	0	0	0
45	0	1	1	0	0	1	0	0	0	1	1	0	0	0	0	0
46	0	1	1	0	0	1	0	0	0	1	0	1	0	0	0	0
47	0	1	1	0	0	1	0	0	0	1	0	1	0	0	0	0
48	0	1	1	0	0	1	0	0	0	1	0	1	0	0	0	0
49	0	1	1	0	0	1	0	0	0	1	0	1	0	0	0	0
50	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
51	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
52	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
53	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	0
54	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	0
55	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
56	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0
57	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0
58	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0
59	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0
60	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
61	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Table 5.12: Entity-attribute matrix for the source code in the Figure 5.23

Table 5.13 shows the output of A-KNN for K=3. In each step, the algorithm will merge the most similar two clusters to form a new cluster, and give the new cluster a unique ID.

Step	1st cluster	2nd cluster	New cluster	Dissimilarity
1	0	1	62	0
2	2	3	63	0
3	63	4	64	0
4	64	5	65	0
5	65	6	66	0
6	66	7	67	0
7	67	8	68	0
8	9	10	69	0
9	69	11	70	0
10	70	12	71	0
11	71	13	72	0
12	72	14	73	0
13	73	15	74	0
14	74	16	75	0
15	75	17	76	0
16	76	18	77	0
17	77	19	78	0
18	78	24	79	0
19	79	25	80	0
20	80	26	81	0
21	81	27	82	0
22	82	28	83	0
23	83	29	84	0
24	84	30	85	0
25	85	50	86	0
26	86	51	87	0
27	87	52	88	0
28	88	55	89	0
29	20	21	90	0
30	31	32	91	0
31	33	34	92	0
32	92	35	93	0
33	93	36	94	0
34	37	38	95	0
35	95	39	96	0
36	76	40	97	0
37	42	43	98	0
38	98	44	99	0
39	99	45	100	0
40	46	47	101	0
41	101	48	102	0

42	102	49	103	0
43	53	54	104	0
44	56	57	105	0
45	58	59	106	0
46	60	61	107	0
47	92	37	108	0
48	41	42	109	0.8
49	109	46	110	0.8
50	20	22	111	0.75
51	111	23	112	0.75
52	31	41	113	0.75
53	9	20	114	0.67
54	114	31	115	0.67
55	115	53	116	0.67
56	2	116	117	0.5
57	117	56	118	0.5
58	118	58	119	0.5
59	119	92	120	0.4
60	62	63	121	0.27

Table 5.13: Output of A-KNN (K=3) for the source code in the Figure 5.23

The results provide several suggestions for partitioning of the code according the levels on the hierarchy. At the low level of hierarchy, we will get many functions with few lines of code. At the high levels of the hierarchy, we will get few functions with a large number of lines of code. According to this result, the function can be divided into parts as Figure 5.24 shows.

```
public void execute(AbstractParsedCommand parsedCommand) throws ConsoleException
{
```

```
0 if((parsedCommand != null) && (parsedCommand instanceof MixParsedCommand)){
1 MixParsedCommand inputCommand = (MixParsedCommand) parsedCommand;
```

Block1

```
2       setWorkIndeterminate();
3       Document pdfDocument = null;
4       PdfCopy  pdfWriter = null;
5       PdfReader pdfReader1;
6       PdfReader pdfReader2;
7       int[] limits1 = {1,1};
8        int[] limits2 = {1,1};
```

Block 2

```
9  try{
10      File tmpFile = FileUtility.generateTmpFile(inputCommand.getOutputFile());
11      pdfReader1 = new PdfReader(new
        RandomAccessFileOrArray(inputCommand.getFirstInputFile().getFile().getAbsoluteP
        ath()),inputCommand.getFirstInputFile().getPasswordBytes());
12      pdfReader1.consolidateNamedDestinations();
13      limits1[1] = pdfReader1.getNumberOfPages();
14      pdfReader2 = new PdfReader(new
        RandomAccessFileOrArray(inputCommand.getSecondInputFile().getFile().getAbsolute
        Path()),inputCommand.getSecondInputFile().getPasswordBytes());
15      pdfReader2.consolidateNamedDestinations();
16      limits2[1] = pdfReader2.getNumberOfPages();
17      pdfDocument = new Document(pdfReader1.getPageSizeWithRotation(1));
18      log.debug("Creating a new document.");
19      pdfWriter = new PdfCopy(pdfDocument, new FileOutputStream(tmpFile));
20      if(inputCommand.getOutputPdfVersion() != null){
21              pdfWriter.setPdfVersion(inputCommand.getOutputPdfVersion().charValue());
                }
22      if(inputCommand.isCompress()){
23                      pdfWriter.setFullCompression();
        }
24      pdfDocument.addCreator(ConsoleServicesFacade.CREATOR);
25      pdfDocument.open();
26      PdfImportedPage page;
27      boolean finished1 = false;
28      boolean finished2 = false;
29      int current1 = (inputCommand.isReverseFirst())? limits1[1] :limits1[0];
30      int current2 = (inputCommand.isReverseSecond())? limits2[1] :limits2[0];
50      pdfWriter.freeReader(pdfReader1);
51      pdfWriter.freeReader(pdfReader2);
52      pdfDocument.close();
53      if(FileUtility.renameTemporaryFile(tmpFile, inputCommand.getOutputFile(),
        inputCommand.isOverwrite())){
54                      log.debug("File
        "+inputCommand.getOutputFile().getCanonicalPath()+" created.");
        }
55      log.info("Alternate mix completed.");
31      while(!finished1 || !finished2){
41              if(!finished2){
32                      if(!finished1){
```

Block3

```
33  if(current1>=limits1[0] && current1<=limits1[1]){
34             page = pdfWriter.getImportedPage(pdfReader1, current1);
35             pdfWriter.addPage(page);
36             current1 = (inputCommand.isReverseFirst())? (current1-1) :(current1+1);
}
```

Block 4

```
37 else{
38      log.info("First file processed.");
39      pdfReader1.close();
40      finished1 = true;
        }
        }
```

Block 5

```
42      if(current2>=limits2[0] && current2<=limits2[1] && !finished2){
43      page = pdfWriter.getImportedPage(pdfReader2, current2);
44      pdfWriter.addPage(page);
45      current2 = (inputCommand.isReverseSecond())? (current2-1) :(current2+1);
```

Block 6

```
46      }else{
47              log.info("Second file processed.");
48              pdfReader2.close();
49              finished2 = true;
        }
        }
        }
```

Block 7

```
56      }catch(Exception e){
57              throw new MixException(e);
58      }finally{
59              setWorkCompleted();
        }
```

Block 8

```
60      }else{
61              throw new ConsoleException(ConsoleException.ERR_BAD_COMMAND);
        }
        }
```

Block 9

Figure 5.24: Blocks [1-9], one of the suggested solutions by A-KNN to do refactoring

Thus, the function *execute* can be partitioned into nine functions and one more function that contains invocations of the nine functions. Overall, instead of having one low-cohesive function, we can use ten high-cohesive functions. Figure 5.25 shows the containing class of the method *executes* before refactoring. Figure 5.26 shows the containing class of the method *executes* after refactoring. The class in Figure 5.26 is better than the class in Figure 5.25 because it is more understandable and it contains functions with smaller size which is easier to maintain.

```
package org.pdfsam.console.business.pdf.handlers;
import java.io.File;
import java.io.FileOutputStream;
import org.apache.log4j.Logger;
import org.pdfsam.console.business.ConsoleServicesFacade;
import org.pdfsam.console.business.dto.commands.AbstractParsedCommand;
import org.pdfsam.console.business.dto.commands.MixParsedCommand;
import org.pdfsam.console.business.pdf.handlers.interfaces.AbstractCmdExecutor;
import org.pdfsam.console.exceptions.console.ConsoleException;
import org.pdfsam.console.exceptions.console.MixException;
import org.pdfsam.console.utils.FileUtility;
import com.lowagie.text.Document;
import com.lowagie.text.pdf.PdfCopy;
import com.lowagie.text.pdf.PdfImportedPage;
import com.lowagie.text.pdf.PdfReader;
import com.lowagie.text.pdf.RandomAccessFileOrArray;
/**
 * Command executor for the alternate mix command
 * @author Andrea Vacondio
 */
public class AlternateMixCmdExecutor extends AbstractCmdExecutor{


private final Logger log =
Logger.getLogger(AlternateMixCmdExecutor.class.getPackage().getName());
public void execute(AbstractParsedCommand parsedCommand) throws ConsoleException {
0 if((parsedCommand != null) && (parsedCommand instanceof MixParsedCommand)){
1       MixParsedCommand inputCommand = (MixParsedCommand) parsedCommand;
2       setWorkIndeterminate();
3       Document pdfDocument = null;
4       PdfCopy  pdfWriter = null;
5       PdfReader pdfReader1;
6       PdfReader pdfReader2;
7       int[] limits1 = {1,1};
8       int[] limits2 = {1,1};
9       try{
10      File tmpFile = FileUtility.generateTmpFile(inputCommand.getOutputFile());
```

```
11      pdfReader1 = new PdfReader(new
        RandomAccessFileOrArray(inputCommand.getFirstInputFile().getFile().getAbsolutePath(
        )),inputCommand.getFirstInputFile().getPasswordBytes());
12              pdfReader1.consolidateNamedDestinations();
13              limits1[1] = pdfReader1.getNumberOfPages();
14              pdfReader2 = new PdfReader(new
                RandomAccessFileOrArray(inputCommand.getSecondInputFile().getFile().getAbso
                lutePath()),inputCommand.getSecondInputFile().getPasswordBytes());
15              pdfReader2.consolidateNamedDestinations();
16              limits2[1] = pdfReader2.getNumberOfPages();
17              pdfDocument = new Document(pdfReader1.getPageSizeWithRotation(1));
18              log.debug("Creating a new document.");
19              pdfWriter = new PdfCopy(pdfDocument, new FileOutputStream(tmpFile));
20              if(inputCommand.getOutputPdfVersion() != null){
21              pdfWriter.setPdfVersion(inputCommand.getOutputPdfVersion().charValue());
                }
22              if(inputCommand.isCompress()){
23                      pdfWriter.setFullCompression();
                }
24              pdfDocument.addCreator(ConsoleServicesFacade.CREATOR);
25              pdfDocument.open();
26              PdfImportedPage page;
27              boolean finished1 = false;
28              boolean finished2 = false;
29              int current1 = (inputCommand.isReverseFirst())? limits1[1] :limits1[0];
30              int current2 = (inputCommand.isReverseSecond())? limits2[1] :limits2[0];
31              while(!finished1 || !finished2){
32                      if(!finished1){
33                              if(current1>=limits1[0] && current1<=limits1[1]){
34                              page = pdfWriter.getImportedPage(pdfReader1, current1);
35                              pdfWriter.addPage(page);
36              current1 = (inputCommand.isReverseFirst())? (current1-1) :(current1+1);
37                      }else{
38                              log.info("First file processed.");
39                              pdfReader1.close();
40                              finished1 = true;
                        }
                        }
41                      if(!finished2){
42              if(current2>=limits2[0] && current2<=limits2[1] && !finished2){
43                              page = pdfWriter.getImportedPage(pdfReader2, current2);
44                              pdfWriter.addPage(page);
45              current2 = (inputCommand.isReverseSecond())? (current2-1) :(current2+1);
46                      }else{
47                              log.info("Second file processed.");
48                              pdfReader2.close();
49                              finished2 = true;
                        }
                        }
                }
50              pdfWriter.freeReader(pdfReader1);
51              pdfWriter.freeReader(pdfReader2);
52              pdfDocument.close();
53              if(FileUtility.renameTemporaryFile(tmpFile,
                inputCommand.getOutputFile(),inputCommand.isOverwrite())){
54              log.debug("File "+inputCommand.getOutputFile().getCanonicalPath()+"
                created.");
               }
55              log.info("Alternate mix completed.");
56              }catch(Exception e){
57                      throw new MixException(e);
58              }finally{
59                      setWorkCompleted();
                }
60              }else{
61                      throw new ConsoleException(ConsoleException.ERR_BAD_COMMAND);
                }

        }

}
```

Figure 5.25: Containing class of the method *executes* before refactoring

```
package org.pdfsam.console.business.pdf.handlers;

import java.io.File;
```

```
import java.io.FileOutputStream;
import org.apache.log4j.Logger;
import org.pdfsam.console.business.ConsoleServicesFacade;
import org.pdfsam.console.business.dto.commands.AbstractParsedCommand;
import org.pdfsam.console.business.dto.commands.MixParsedCommand;
import org.pdfsam.console.business.pdf.handlers.interfaces.AbstractCmdExecutor;
import org.pdfsam.console.exceptions.console.ConsoleException;
import org.pdfsam.console.exceptions.console.MixException;
import org.pdfsam.console.utils.FileUtility;
import com.lowagie.text.Document;
import com.lowagie.text.pdf.PdfCopy;
import com.lowagie.text.pdf.PdfImportedPage;
import com.lowagie.text.pdf.PdfReader;
import com.lowagie.text.pdf.RandomAccessFileOrArray;
/**
 * Command executor for the alternate mix command
 * @author Andrea Vacondio
 */
public class AlternateMixCmdExecutor extends AbstractCmdExecutor{

private final Logger log =
Logger.getLogger(AlternateMixCmdExecutor.class.getPackage().getName());
Document pdfDocument;
PdfCopy  pdfWriter;
PdfReader pdfReader1;
PdfReader pdfReader2;
int[] limits1 = new int[2];
int[] limits2 = new int [2]
PdfImportedPage page;
MixParsedCommand inputCommand;
boolean finished1;
boolean finished2;
int current1;
int current2;

public void initialize(AbstractParsedCommand parsedCommand){
inputCommand = (MixParsedCommand) parsedCommand;
setWorkIndeterminate();
Document pdfDocument = null;
PdfCopy  pdfWriter = null;
limits1[0]=1
limits1[1]=1;
limits2[0]=1;
limits2[1]=1;
finished1 = false;
finished2 = false;
}

public void finishProcess1(){
page = pdfWriter.getImportedPage(pdfReader1, current1);
pdfWriter.addPage(page);
current1 = (inputCommand.isReverseFirst())? (current1-1) :(current1+1);
}

public void closeProcess1(){
log.info("First file processed.");
pdfReader1.close();
finished1 = true;
}

public void finishProcess2(int current2){
page = pdfWriter.getImportedPage(pdfReader2, current2);
pdfWriter.addPage(page);
current2 = (inputCommand.isReverseSecond())? (current2-1) :(current2+1);
}

public void closeProcess2(){
log.info("Second file processed.");
pdfReader2.close();
finished2 = true;
}

public void endProcess() throws ConsoleException(){
throw new ConsoleException(ConsoleException.ERR_BAD_COMMAND);
}

public void runExecution() {
```

```
try{
File tmpFile = FileUtility.generateTmpFile(inputCommand.getOutputFile());
 pdfReader1 = new PdfReader(new
RandomAccessFileOrArray(inputCommand.getFirstInputFile().getFile().getAbsolutePath()),inpu
tCommand.getFirstInputFile().getPasswordBytes());
pdfReader1.consolidateNamedDestinations();
limits1[1] = pdfReader1.getNumberOfPages();
pdfReader2 = new PdfReader(new
RandomAccessFileOrArray(inputCommand.getSecondInputFile().getFile().getAbsolutePath()),inp
utCommand.getSecondInputFile().getPasswordBytes());
pdfReader2.consolidateNamedDestinations();
limits2[1] = pdfReader2.getNumberOfPages();
pdfDocument = new Document(pdfReader1.getPageSizeWithRotation(1));
log.debug("Creating a new document.");
pdfWriter = new PdfCopy(pdfDocument, new FileOutputStream(tmpFile));
if(inputCommand.getOutputPdfVersion() != null){
pdfWriter.setPdfVersion(inputCommand.getOutputPdfVersion().charValue());
}
if(inputCommand.isCompress()){
pdfWriter.setFullCompression();
}
pdfDocument.addCreator(ConsoleServicesFacade.CREATOR);
pdfDocument.open();
current1 = (inputCommand.isReverseFirst())? limits1[1] :limits1[0];
current2 = (inputCommand.isReverseSecond())? limits2[1] :limits2[0];
pdfWriter.freeReader(pdfReader1);
pdfWriter.freeReader(pdfReader2);
pdfDocument.close();
if(FileUtility.renameTemporaryFile(tmpFile, inputCommand.getOutputFile(),
inputCommand.isOverwrite())){
log.debug("File "+inputCommand.getOutputFile().getCanonicalPath()+" created.");
}
log.info("Alternate mix completed.");
while(!finished1 || !finished2){
if(!finished2){
if(!finished1){
if(current1>=limits1[0] && current1<=limits1[1])
finishProcess1();
else
loseProcess1();
if(current2>=limits2[0] && current2<=limits2[1] && !finished2)
finishProcess2();
else
closeProcess2();
}
}
}
}
catch(Exception e){
throw new MixException(e);
}
finally{
setWorkCompleted();
}
}

public void execute(AbstractParsedCommand parsedCommand){
if((parsedCommand != null) && (parsedCommand instanceof MixParsedCommand)){
initialize(parsedCommand);
runExecution();
}
else
endProcess();
}
}
```

Figure 5.26: Containing class of the method *executes* after refactoring

Although the class has more lines of code after refactoring than before refactoring, it is accepted in software engineering that a class with small understandable functions is better

than a class with long functions [25]. Consequently, A-KNN showed an excellent performance. This algorithm proved to give useful suggestions which aid the software designer in refactoring. The implemented A-KNN has an advantage over previous clustering techniques (SLINK, CLINK and WPGMA) as it reduces the amount of computations needed. This is derived by from calculating the distances between classes once in A-KNN. Calculating the distances between the clusters was consuming an excessive amount of computation in the three previous algorithms. A-KNN finds the most similar three pairs of clusters, and it determines which two clusters are to be merged without recalculating all of the cluster's similarities. Other techniques (SLINK, CLINK, WPGMA) recalculate the similarities after each iteration. Thus, instead of calculating the distance between two clusters, each consisting of many entities (at least two entities), A-KNN depends on the distance between the entities of the cluster individually. In other words, A-KNN uses clusters consisting of one element. Consequently, the amount of computation in each clustering step is less than for SLINK, CLINK and WPGMA.

5.9. Conclusions

We investigated software refactoring by clustering. Specifically, we investigated the selection of entities and attributes, software metrics, similarity measures, resemblance coefficient experiments, hierarchical agglomerative algorithms, and the application of the approach to a source code. A comparative study was conducted to choose the best clustering technique and the most suitable similarity measure weight. We showed that WPGMA is better than SLINK and CLINK. The weight 8:3 was the best for the previous

three algorithms. Then we compared our new implemented algorithm A-KNN with the previous three algorithms. The experimental results showed that A-KNN performs competitively with the other algorithms, and in some cases it is better than all of them. In addition, it requires fewer computational operations.

CHAPTER 6

SOFTWARE REFACTORING AT THE CLASS LEVEL

6.1. Introduction

In the previous chapter we applied clustering at the function level to assist software designers in refactoring. In addition, we proposed a new clustering technique using adaptive K-Nearest Neighbor which was applied at the function level. It was proven to be more efficient than published work in the literature. In this chapter, we are addressing software refactoring at the class level using pattern recognition techniques. In addition, we are proposing two approaches to achieve the balance between coupling and cohesion at the class level. The first approach achieves this balance by suggesting the moving of methods from one class to another by using clustering techniques. Thus, the proposed approach enhances the structure of the classes by presenting guide-lines to shift methods among classes. In other words, the approach defines the ill-structured classes and gives advice on correcting the structure-errors in those classes. While the first approach keeps the number of classes fixed, the second approach provides a mechanism to find the best

number of the classes to use. The second approach provides a new design for the classes. Hence, the number of the classes in the new design is not affected by the number of classes in the original system. In other words, instead of correcting the current errors in the original system, the approach suggests a complete new design for the system classes.

Due to this main difference between the two approaches, the first proposed approach called “software refactoring at class level using clustering with fixed number of classes”. All the methods in the system’s classes will be assigned to one of the classes. In this way, the size of the class grows by adding new methods to the cluster center representing the class. The methods which are added first to one class are more similar to the class than those which are added later. Thus, at the base of the class cluster, the methods are more similar to the class than the methods at the top of the class cluster. At the end, each of these clusters will be mapped to a complete class in the system. The grouped methods around the cluster center form the functionality of the class.

The second approach is called “software refactoring at class level using clustering with variable number of classes”. The first approach uses a fixed number of classes and it groups the methods around these class clusters according to their similarity with the cluster center. The second approach groups the methods into class clusters based on the similarity of these methods with each other. Similar methods are grouped in the same class. In adding a method to a class, the similarity between the method and all other methods in the cluster is evaluated. Based on this evaluation, a decision is taken to add the method to one of the classes. Since similar methods will be grouped in the same cluster,

then this similarity between methods determines the number of clusters in system. Consequently, if the similarity among all methods is large, then a small number of classes will be needed. On the other hand, if the similarity among the methods is small, then a large number of classes will be needed.

After explaining the approaches' input, output, mechanism, controllers and settings, we will describe a set of experimental studies which were conducted to test the effectiveness of the suggested approaches.

The rest of this chapter is organized as follows. Section 2 describes "software refactoring at class level with fixed number of classes". Section 3 describes "software refactoring at class level using variable number of classes". Section 4 describes the experimental results. Finally section 5 presents the conclusions.

6.2. Software Refactoring at the Class Level Using Clustering With Fixed Number of Classes

Our approach is based on our adaptive clustering technique. The entities to be clustered are the classes' methods, and the number of clusters is taken as the number of classes. Thus, if we have n classes with m methods, then we will have n cluster centers and m methods for clustering. In other words, each method will be assigned to one of the n centers. The assigning process is based on the similarity measure used. The similarity measure between the method and a class is taken as the number of attributes that the method uses from that class. For instance, if a method m_1 uses three attributes of Class A

and two attributes of Class B, then method m_1 is more similar to Class A than to Class B. This means that method m_1 will be assigned to the cluster center representing Class A. The algorithm is described in the following steps:

1) Calculate the similarity between the methods and the containing classes (cohesion).
2) Calculate the similarity between all the methods and other classes (coupling).
3) Find the maximum similarity for each method.
4) Move the method to the class with maximum similarity.

Using this approach, each method will be assigned to the class with more attributes that the method uses.

The suggested clustering mechanism depends on the similarity measure. Thus, if we have n classes and m methods in those classes, then we will obtain m × n similarity matrix. The rows represent the methods, and columns represent the classes (cluster centers), and the values in the array represent the similarity measure values between the methods and classes. The similarity value between class and method is the number of the class attributes that the method uses. For each method, there will be a row in the similarity matrix. The values in this row represent the similarity between the method and all other classes:

$$Sim(m_i, C_j) = \text{No of the } C_i\text{'s attributes which the method uses.}$$

$$\text{Where: } 1 \leq i \leq n,\ 1 \leq j \leq m$$

In reality, the values of this matrix are indirectly representing the coupling and cohesion values. For example, if m =16 and n=4, then we have 16 methods to be assigned to 4 classes centers. Suppose a method m_5 belongs to class 2. Then, in the similarity matrix,

the element $el_{[5,2]}$ represents the cohesion values (the number of attributes that the method uses from its own class). The elements $el_{[5,1]}$, $el_{[5,3]}$, $el_{[5,4]}$ in row 5 represent the coupling values with classes 1, 3, 4, respectively, (i.e. they are the number of attributes which the method accesses from those classes). In this way, the mechanism works on the actual values of coupling and cohesion, in order to reduce the coupling between classes and to increase the cohesion so as to achieve a balance between the two.

6.3. Software Refactoring at the Class Level Using Clustering With Variable Number of Classes

In the previous sections, we presented a clustering approach at the class level. The presented approach redistributes the methods over the original system classes. Thus, the presented approach aims to balance the coupling and cohesion of the code without using the same number of classes. Consequently, the technique does not suggest forming new classes. In the following sections, we are presenting an approach to suggest a new arrangement for the classes and to redistribute the methods in the new classes. The clustering technique suggests different designs for the class's structure, so that the software designer can choose the best design for the current system. The effects of this approach on the code are investigated by using two software metrics.

6.3.1. The approach

This section provides an approach to classes refactoring using pattern recognition techniques, and it discusses the clustering technique schema.

6.3.1.1. Classes Refactoring Approach

The aim of the classes refactoring is to enhance the structure of the system. We developed a set of tools to help us in this process. After clustering, the system will be provided in a tree structure. The approach provides information about the existing structure of the classes and heuristic guide-lines to improve the current structure. These guide-lines can be used to aid the software designer in refactoring the current system. Figure 6.1 shows the flow of the approach.

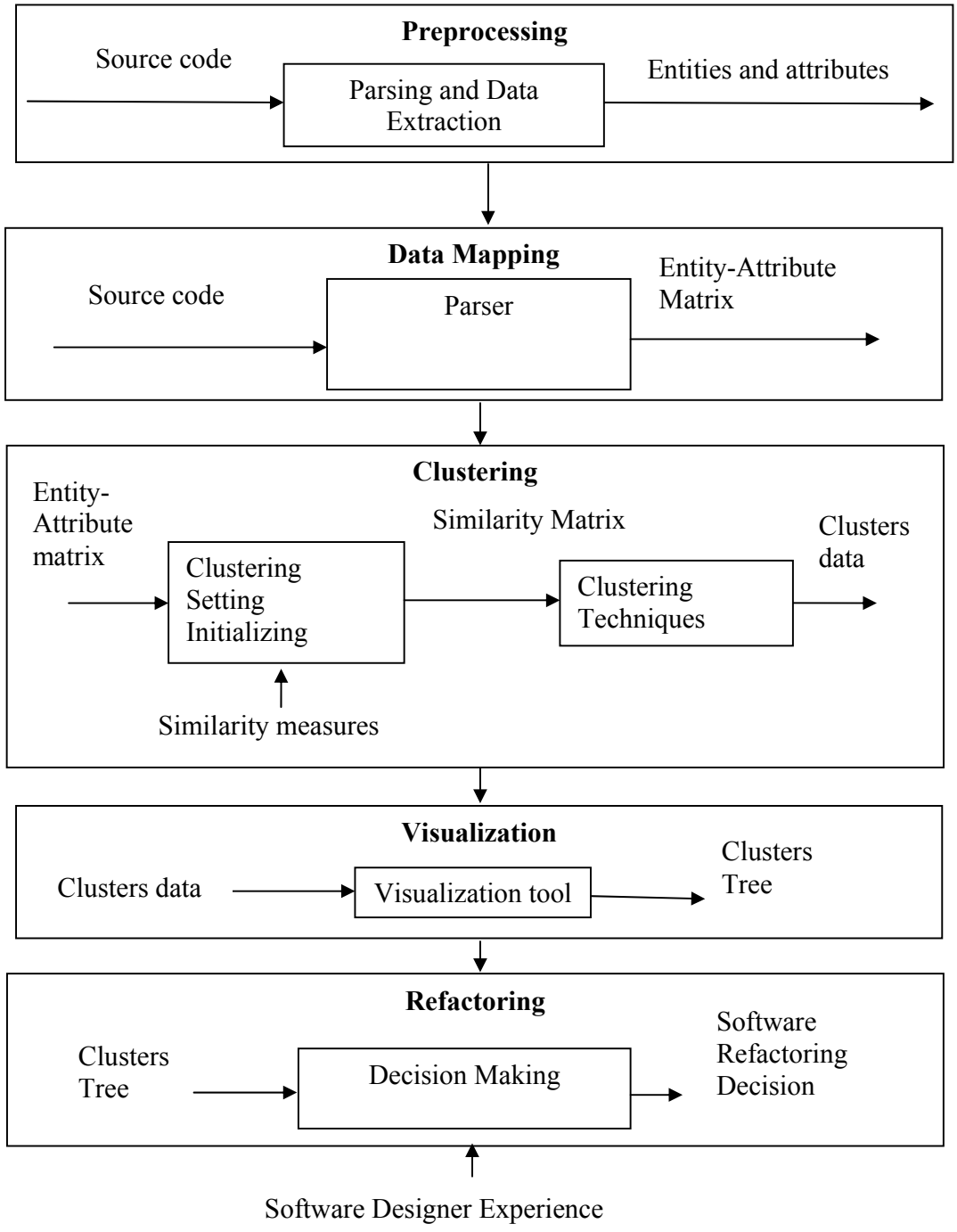


Figure 6.1: Approach to classes refactoring

The approach consists of five phases:

1. Preprocessing: In this phase, the source code is parsed, and a list of entities and attributes is extracted.

2. Data Mapping: In this phase the entity-attribute matrix will be filled. Each entity may contain one or more attributes. Entities are the items that will be grouped by the attributes they share. Thus, the more attributes two entities share, the more related those entities are.

3. Clustering: The controllers of this phase are the similarity measures. A metric called “resemblance coefficient” is calculated to measure the similarity between every two entities. Mainly, the similarity is measured by the common attributes that two entities share. After the resemblance coefficient is calculated, the clusters are formed by using the clustering techniques (SLINK, CLINK, and WPGMA) and the A-KNN clustering algorithm.

4. Visualization: The results of the previous phase are displayed as a tree structure. The tree shows the current structure of the classes. Closely related entities are grouped in the same class cluster.

5. Refactoring: In this phase, the clustering tree provides heuristic advice on how to refactor the code. However, software designers’ experience plays an essential role to make the final design decision according to the refactoring objectives. The approach is able to generate clusters automatically.

6.3.1.2. Entities

Entities are those items that need to be grouped. For software refactoring at the class level, methods were chosen as entities. This is because methods are the basic elements of classes. Class constructor was considered as any other method inside the class, because this constructor may contain many initialization statements to the data-members inside the class. Moreover, sometimes class constructor may contain invocations to other methods inside the class.

6.3.1.3. Attributes

The features of entities are called attributes. There may be many attributes for each entity. The attributes of each entity can describe its different properties. Based on this, entities will be more similar if they share many common attributes. The attributes will be used to calculate how closely two entities are. To do clustering at the class level, the possible attributes for the entities should be identified. Different class data-members may be related to different functional tasks. Therefore, the class data-members are used as attributes for the entities. An entity has access to all data-members inside the class. It can also access data-members of other classes by using instances from those classes. However, the instances from other classes inside the class were not considered as attributes for the entities. This is because there is no interest in the use of the instance itself. We are interested in how the method uses the data-member from the other classes via this instance. Based on this, the attribute is a data-member with a predefined data type (int, double, String……) attached with the label of the containing class. Thus, each attribute is measured as a quantitative scale representation. The attribute value is the

number of times the method access as this data-member represented by this attribute. The method can access data-member directly or indirectly. In the direct manner, the method accesses data-members in the containing class or in other classes (by using instances of those classes). The method can access a data-member indirectly by invoking other methods which use this data-member. Thus, the value of the attribute can be measured by using the following formula:

$$V(att_i, ent_j) = da(att_i, ent_j) + ia(att_i, ent_j);\ i \in [1..a], j \in [1..b]$$

where:

$V(att_i, ent_j)$: is the value of the attribute att_i for the entity ent_j

a: is the number of the attributes in the system

b: is the number of the methods in the system

da: is the number of the direct accesses to the data-member represented by the attribute att_i in the method represented by the entity ent_j

ia: is the number of indirect accesses to the data-member represented by the attribute att_i in the method represented by the entity ent_j

$$ia(att_i, ent_j) = \sum_{k=0}^{t} m_k n_k; k \neq j$$

where:

t : is the number of the other entities that are called inside the entity ent_j

m_k: is the number of the invocations to entity ent_k inside entity ent_j

$n_k = da(att_i, ent_k)$: the number of direct accesses to the data-member represented by attribute att_j in the method represented by entity ent_k

Based on the previous discussion, the entities-attributes matrix will be filled. For any two entities in the entities-attribute matrix, there are three types of matches:

0-n/n-0 match: the attribute is present n times in one entity, and it is absent from the other entity (n>0)

n-m match: the attribute is present n times in one entity and m times in the other entity (n>0 and m>0).

0-0 match: the attribute is absent from both entities.

6.3.1.4. Similarity Measure

To evaluate the closeness between two entities, we need to use a similarity measure. The similarity measure is represented by a resemblance coefficient. This coefficient depends mainly on the entities and their attributes.

6.3.1.4.1. Attributes

The more attributes two entities share, the more similar they are. If the entities share many attributes, this will indicate that those two entities have closely related functionalities. Thus, from the cohesion point-of-view, they should be kept in the same class.

6.3.1.4.2. Matches

A match can be one of three types: n-0/0-n, n-m, and 0-0. The first type means a mismatch between two entities. This provides a negative contribution in the coefficient resemblance. In other words, this is an indication of dissimilarity between the two entities. Therefore, this type of match was considered in the resemblance coefficient. Matches of

the second type n-m mean that two entities share at least one attribute (m>0 and n>0). This provides a positive contribution to the resemblance coefficient. Therefore, n-m matches were considered in the similarity coefficient. The last type (0-0) means a mismatch (i.e. attribute is not accessed by any of the two entities). There is only one entity-attribute matrix for a system. The columns represent all the attributes in the system. The attributes are the classes' data-members from predefined types (int, double, String……) and each attribute is labeled with the class that contains that attribute. The rows represent all the methods in the classes. Since the columns represent all attributes in the system, it is expected to have many mismatches (0-0 matches) in the matrix. This is because each method uses a small number of attributes. Several studies in restructuring showed that better results could be obtained by ignoring 0-0 matches [31, 32, 56]. These matches will generate a distortion in the similarity measure. Therefore, these matches were not considered in the similarity coefficient.

6.3.1.4.3. Resemblance coefficient

Based on the previous considerations, the resemblance coefficient between two entities is given by the following formula:

$$Coeff = \frac{similarity\ factor}{similarity\ factor + dissimilarity\ factor}$$

Coeff: resemblance coefficient

$$\text{similarity factor} = \sum_{k=0}^{a} \min (n_k, m_k)$$

where:

a: is the number of n-m matches between two entities

dissimilarity factor= number of n-0/0-n matches between two entities

Therefore, if no common attribute is shared by two entities, the Coeff will be 0. If there are no n-0 matches between two entities, the Coeff will be 1. The values of Coeff will be between 0 and 1. However, in the real implementation, we used the complement of Coeff which is given by following formula:

Diss-Coeff= 1 - Coeff

where:

Diss-Coeff is the dissimilarity coefficient

This was done only to provide more flexibility in the implementation, and it has no effect in the results.

6.4. Experimental Results

We conducted a set of experiments to test the effectiveness of the presented approaches. The experiments were conducted on different experimental units from different sources. The first approach (software refactoring at class level using clustering with fixed number of classes) uses a different mechanism in clustering from the approaches presented in chapter 5. The approaches presented in chapter 5 (which work at the function level) use either traditional clustering algorithms like SLINK, CLINK and WPGMA or our new proposed algorithm A-KNN. Software refactoring at class level using clustering with

fixed number of classes uses neither one of the traditional clustering algorithm SLINK, CLINK and WPGMA, nor our new proposed algorithm A-KNN. It uses an introduced mechanism in which methods are grouped according to their similarity with the classes, not with each other. Due to this, it was necessary to test the approach in a test-code before testing the code in an industrial system. Testing the approach in a test-code is a short-term process which enables us to decide whether the approach can help to achieve the purpose. The advantage of the test-code over the industrial system is that the test-code is written in such a way as to test all the possible cases that may affect the performance of the mechanism. The industrial system may not cover all the cases which might affect the performance of the mechanism. For instance, the test-code is able to cover all types of communications between classes, while the industrial system may not do that. The mechanism is also tested in the industrial system. The second proposed approach in this chapter (software refactoring at class level using clustering with variable number of classes) uses the same algorithms used by the approaches in chapter 5. Thus, there was no need to test the approach on a test-code, and the approach was tested immediately on industrial systems. The following subsections describe the setting of the experiments (in the two approaches) and the results of these experiments.

6.4.1. Experimental Results on Software Refactoring at Class Level Using Clustering With Fixed Number of Classes

In the next subsections we present the results of applying the approach on different source codes. We started by applying the code on a test source code. Then, we applied it on an

industrial system. After each experiment, we analyzed the results to see the effect of the approach.

6.4.1.1. Experimental Results on Test Source Code

We applied the previous approach on the source code in Figure 6.2. In this source code, we have 4 classes A, B, C and D and 16 methods {method0, method1, ……., method15}. The following source code shows the classes, structure before clustering. This code was written just to test the technique. We did not include the set(s) and get(s) methods of the attributes. However, this will not affect the transparency of our results.

```
package refactoring;
public class A {
  int a_attr1, a_attr2, a_attr3, a_attr4;
  B b1=new B();
  public A() {
  }
  public int method0(){
    return a_attr1+
b1.b_attr1+b1.b_attr2;
  }
  public void method1(){
    this.a_attr1=1;
    this.a_attr2=2;
  }
  public void method2(){
    this.a_attr3=7;
    this.a_attr4=8;
  }
  public void method3(){
    this.a_attr1=0;
    this.a_attr2=0;
    this.a_attr3=0;
    this.a_attr4=0;
  }
}
```

(a)

```
package refactoring;
public class B {
  int b_attr1, b_attr2, b_attr3, b_attr4;
  C c1=new C();
  D d1=new D();
  public B() {
  }
  public void method4(){
    this.b_attr1=0;
    this.b_attr2=0;
    this.b_attr3=0;
    this.b_attr4=0;
  }
  public int method5(){
    return b_attr1 + c1.c_attr1 +
d1.d_attr1 + d1.d_attr4;
  }
  public int method6(){
    return this.b_attr4+ this.b_attr3;
  }
  public void method7(){
    d1.d_attr1=7;
    d1.d_attr2=d1.d_attr3+d1.d_attr4;
  }
}
```

(b)

```
package refactoring;
public class C {
  int c_attr1, c_attr2, c_attr3, c_attr4;
   public C() {
  }
  public void method8(){
   this.c_attr1=this.c_attr2+this.c_attr3;
  }
  public void method9()  {
    this.c_attr1=1;
    this.c_attr2=2;
  }
  public void method10(){
    this.c_attr3=7;
    this.c_attr4=8;
  }
  public void method11(){
    this.c_attr1=0;
    this.c_attr2=0;
    this.c_attr3=0;
    this.c_attr4=0;
  }
}
```

(c)

```
package refactoring;
public class D {
int d_attr1, d_attr2, d_attr3, d_attr4;
  A a1=new A();
  B b1=new B();
  C c1=new C();
  public D() {
  }
  public void method12(){
   this.d_attr1 = a1.a_attr2 + a1.a_attr3
+ b1.b_attr1;
  }
  public int method13()  {
    return
c1.c_attr1+c1.c_attr2+c1.c_attr3+c1.c_
attr4;
  }
  public void method14(){
    this.d_attr3=7;
    this.d_attr4=8;
  }
  public void method15(){
    this.d_attr1=0;
    this.d_attr2=0;
    this.d_attr3=0;
    this.d_attr4=0;
  }
}
```

(d)

Figure 6.2: Source code before refactoring (a, b, c, d): the source code of class A, B, C, D, respectively

Table 6.1 shows the similarity matrix for the previous source code. The rows represent the methods to be clustered, and the columns represent the classes. The values inside the table represent the similarity between each method and all other classes in the system. For instance, the similarity of *method0* with class A is 1, because it uses one attribute of this class. The similarity of *method0* with class B is 2, because it uses two instances of class B. The similarity of the same method with classes C and D is 0, because it does not use any attributes of them. The clustering algorithm will go over all the methods from *method0*

until *method15* and assign each method to the nearest class (cluster center). For example, for *method0,* the nearest class (cluster center) is class B, because the similarity between *method0* and Class B has the maximum similarity between *method0* and all other classes. Thus, clusters will be formed gradually around the four cluster centers. In each step of clustering, one of the clusters will be assigned to one of the class cluster centers.

	A	B	C	D
method0	1	2	0	0
method1	2	0	0	0
method2	2	0	0	0
method3	4	0	0	0
method4	0	4	0	0
method5	0	1	1	2
method6	0	2	0	0
method7	0	0	0	4
method8	0	0	3	0
method9	0	0	2	0
method10	0	0	2	0
method11	0	0	4	0
method12	2	1	0	1
method13	0	0	4	0
method14	0	0	0	2
method15	0	0	0	4

Table 6.1: Similarity Matrix for the original source code.

We updated the classes by using the results of the clustering technique. These classes were renamed to UpdatedA, UpdatedB, UpdatedC and UpdatedD. These four new classes have the same attributes, but with different methods distribution. Figure 6.3 shows the source code after clustering.

```
package refactoring;
class UpdatedA {
 int a_attr1, a_attr2, a_attr3, a_attr4;
 D d1=new D();
 B b1=new B();
 public UpdatedA() {
 }
 public void method1(){
        this.a_attr1=1;
        this.a_attr2=2;
 }
public void method2(){
        this.a_attr3=7;
        this.a_attr4=8;
 }
public void method3(){
        this.a_attr1=0;
        this.a_attr2=0;
        this.a_attr3=0;
        this.a_attr4=0;
       }
public void method12(){
         d1.d_attr1 = a_attr2 + a_attr3 +
b1.b_attr1;
        }
}
```

(a)

```
package refactoring;
class UpdatedB {
  int b_attr1, b_attr2, b_attr3,
b_attr4;
  A a1=new A();
  public UpdatedB() {
 }
 public int method0(){
    return a1.a_attr1+
b_attr1+b_attr2;
   }
   public void method4(){
   this.b_attr1=0;
   this.b_attr2=0;
   this.b_attr3=0;
   this.b_attr4=0;
 }
 public int method6(){
   return this.b_attr4+ this.b_attr3;
 }
 }
```

(b)

```
package refactoring;
class UpdatedC {
int c_attr1, c_attr2, c_attr3, c_attr4;
public UpdatedC() {
}
public void method8(){
 this.c_attr1=this.c_attr2+this.c_attr3;
}
public void method9()  {
  this.c_attr1=1;
  this.c_attr2=2;
}
public void method10(){
  this.c_attr3=7;
  this.c_attr4=8;
}
public void method11(){
  this.c_attr1=0;
  this.c_attr2=0;
  this.c_attr3=0;
  this.c_attr4=0;
}
public int method13()  {
   return
c_attr1+c_attr2+c_attr3+c_attr4;
}
}
```

(c)

```
package refactoring;
class UpdatedD {
  int d_attr1, d_attr2, d_attr3, d_attr4;
  B b1=new B();
  C c1=new C();
  public UpdatedD() {
  }
  public int method5(){
     return
b1.b_attr1+c1.c_attr1+d_attr1+d_attr4;
  }
   public void method7(){
  d_attr1=7;
  d_attr2=d_attr3+d_attr4;
  }
  public void method14(){
   this.d_attr3=7;
   this.d_attr4=8;
 }
 public void method15(){
   this.d_attr1=0;
   this.d_attr2=0;
   this.d_attr3=0;
   this.d_attr4=0;
 }
 }
```

(d)

Figure 6.3: Source code before refactoring: (a, b, c, d): the source code of class A, B, C, D, respectively

We calculated the values of cohesion by using LCOM, CTA metrics for the four classes before and after clustering. Table 6.2 and Table 6.3 show the intersection values between methods (how many shared instances between every two methods of the classes), and the LCOM value for each class. Table 6.2 shows the values of LCOM before clustering and Table 6.3 shows the values of LCOM after clustering.

Class A	Class B	Class C	Class D
{method0∩method1}=1 {method0 ∩ method2}=0 {method0 ∩ method3}=1 {method1 ∩ method2}=0 {method1 ∩ method3}=2 {method2 ∩ method3}=2	{method4 ∩ method5}=1 {method4 ∩ method6}=2 {method4 ∩ method7}=0 {method5 ∩ method6}=0 {method5 ∩ method7}=2 {method6 ∩ method7}=0	{method8 ∩ method9}=2 {method8 ∩ method10}=1 {method8 ∩ method11}=3 {method9 ∩ method10}=0 {method9 ∩ method11}=2 {method10 ∩ method11}=2	{method12 ∩ method13}=0 {method12 ∩ method14}=0 {method12 ∩ method15}=1 {method13 ∩ method14}=0 {method13 ∩ method15}=0 {method1 ∩ method15}=2
LCOM= max((2-3),0)=0	LCOM= max((3-3),0)=0	LCOM= max((1-5),0)=0	LCOM= max((4-2),0)=2

Table 6.2: LCOM values before clustering

Class UpdatedA	Class UpdatedB	Class UpdatedC	Class UpdatedD
{method1 ∩method2}=0 {method1 ∩ method3}=2 {method1 ∩ method12}=2 {method2 ∩ method3}=2 {method2 ∩ method12}=1 {method3 ∩ method12}=2	{method0 ∩ method4}=2 {method0 ∩ method6}=0 {method4 ∩ method6}=2	{method8 ∩ method9}=2 {method8 ∩ method10}=1 {method8 ∩ method11}=3 {method8 ∩ method13}=3 {method9 ∩ method10}=0 {method9 ∩ method11}=2 {method9 ∩ method13}=2 {method10 ∩ method11}=2 {method10 ∩ method13}=2 {method11 ∩ method13}=4	{method5 ∩ method7}=2 {method5 ∩ method14}=1 {method5 ∩ method15}=2 {method7 ∩ method15}=4 {method14 ∩ method15}=2
LCOM= max((1-5),0)=0	LCOM=max((1-2),0)=0	LCOM= max((1-9),0)=0	LCOM= max((0-5),0)=0

Table 6.3: Values of LCOM after clustering

Table 6.4 shows the values of CTA before and after refactoring.

Class A	Class B	Class C	Class D
1	2	0	3
2	1	0	0

Table 6.4: Values of CTA before clustering

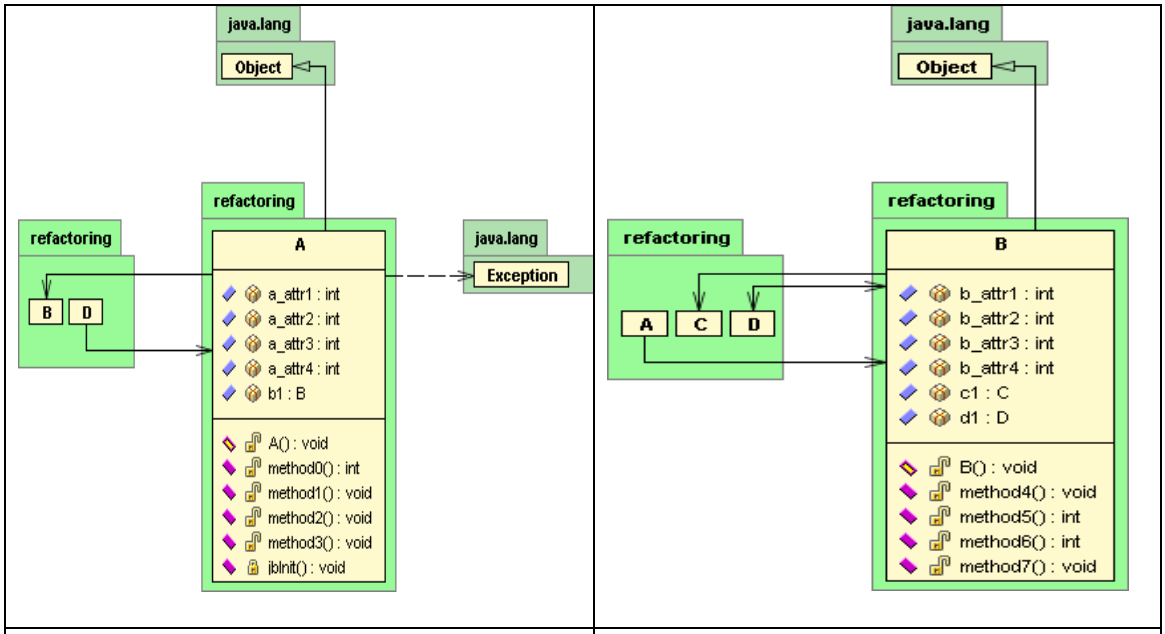
To summarize the results, Table 6.5 shows the changes of LCOM and CTA after the refactoring process.

	A	B	C	D
LCOM	0	0	0	-2
CTA	+1	-1	0	-3

Table 6.5: Change in LCOM and CTA values

From Table 6.5, we can conclude that there is a noticeable decrease in the values of LCOM and CTA. Since the values of LCOM after clustering are less than the values of

LCOM before clustering, the cohesion is better after refactoring than before refactoring. The values of CTA are larger before clustering than after clustering, which means that the coupling has improved after refactoring. This indicates that the previous technique is effective in achieving a balance between coupling and cohesion. Figure 6.4 shows the coupling between classes before clustering.



(a) (b)

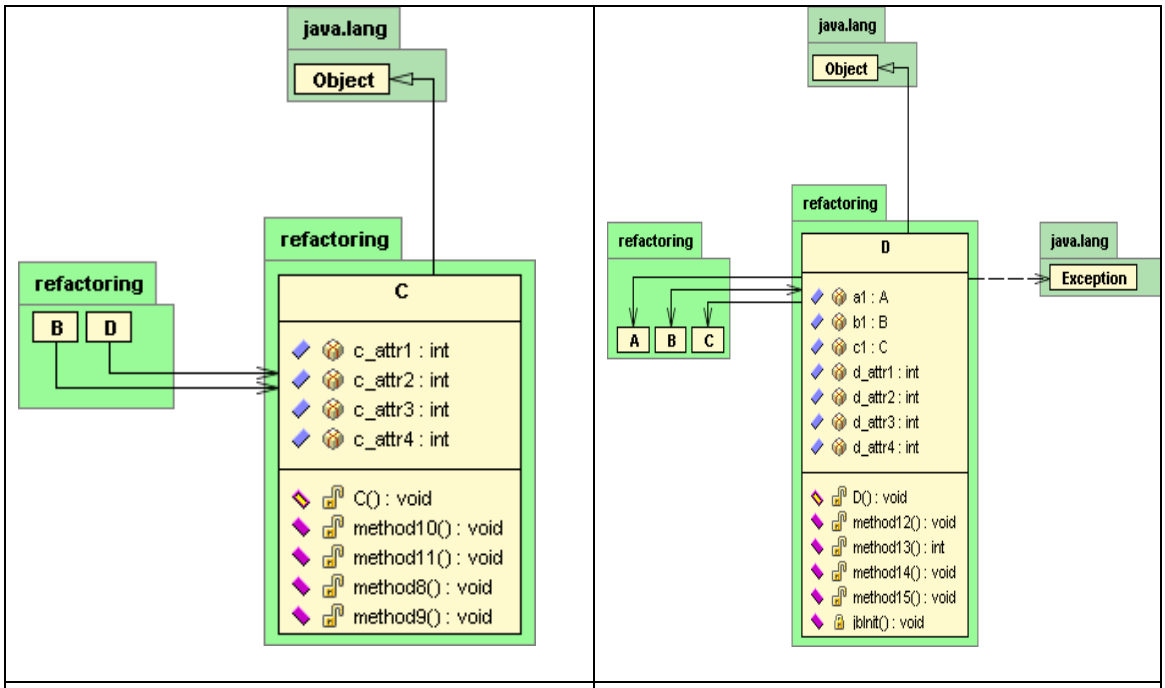


Figure 6.4: Coupling between classes before refactoring: (a, b, c, d): coupling between the class A, B, C, D and the other classes, respectively

Figure 6.5 shows the coupling between classes after clustering. It is clear how the coupling between classes has been decreased after refactoring.

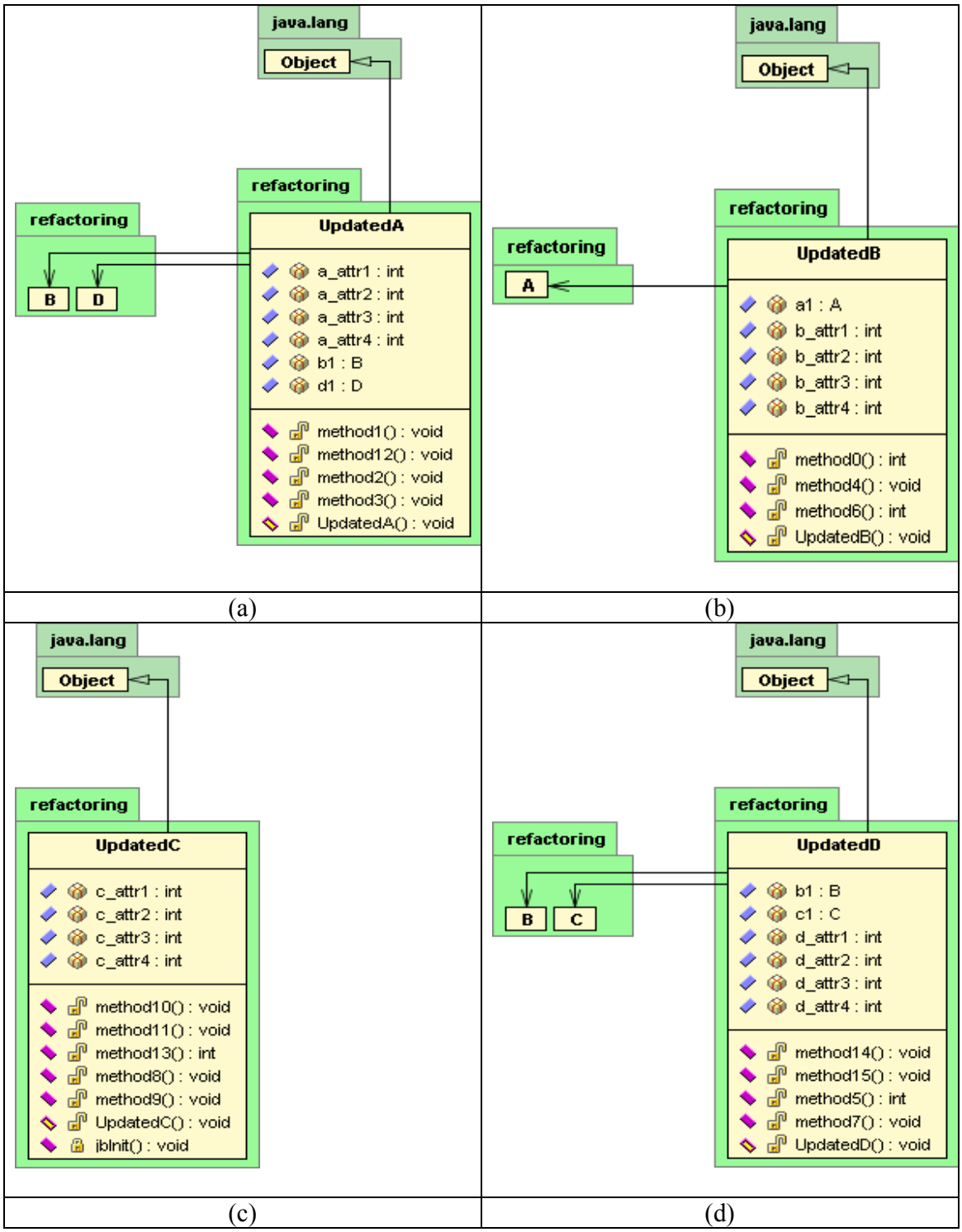


Figure 6.5: Coupling between classes after refactoring: (a, b, c, d): coupling between the class UpdatedA, UpdatedB, UpdatedC, UpdatedD and the other classes, respectively

6.4.1.2. Experimental Results on Industrial System

We applied the technique on industrial software called CSGestionnaire. CSGestionnaire is software developed for French medico-social institutions which helps them to calculate their budgets, prepare and print the invoices for their patients, and much more. It is available at [62]. Table 6.6 shows the similarity matrix for all classes and its methods inside a package called *printing*. It consists of 9 classes and 59 methods.

Method / Class	FacturesEmisesPrinting	Logger	DataProvider	FilteredDataProvider	FacturesPrinting	PreviewPanel	GestionnairePrinting	Printing	FacturesDebiteursPrinting
FacturesEmisesPrinting									
drawFactureHeading	0	0	0	0	0	0	0	0	0
drawOneLineCell	0	0	0	0	0	0	0	0	0
drawTwoLinesCell	0	0	0	0	0	0	0	0	0
drawTableHeader	1	0	0	0	0	0	0	0	0
drawTableCell	0	0	0	0	0	0	0	0	0
drawTableCell	0	0	0	0	0	0	0	0	0
drawTableLine	0	0	0	0	0	0	0	0	0
drawTableSpecialLine	0	0	0	0	0	0	0	0	0
drawFactureTableLines	2	0	0	3	0	0	0	0	0
drawFactureTail	2	0	0	0	0	0	0	0	0
drawFactureTable	0	0	0	0	0	0	0	0	0
drawFootnote	1	0	0	0	0	0	0	0	0
drawFacturePage	0	0	0	0	0	0	0	0	0
Print	0	0	0	0	0	0	0	0	0
getPageCount	1	0	0	0	0	0	0		
FacturesPrinting									
getStrValue	0	0	0	0	0	0	0	0	0
drawEtablissementRect	0	0	0	0	3	0	0	0	0
drawFactureRect	0	0	0	0	3	0	0	0	0
drawReferenceRect	0	0	0	0	3	0	0	0	0
drawDomiciliationBanca	0	0	0	0	3	0	0	0	0

ireRect									
drawResidentRect	0	0	1	0	3	0	0	0	0
drawAssureRect	0	0	1	0	3	0	0	0	0
drawDebiteurRect	0	0	1	0	3	0	0	0	0
drawFactureHeading	0	0	0	0	0	0	0	0	0
drawOneLineCell	0	0	0	0	0	0	0	0	0
drawTwoLinesCell	0	0	0	0	0	0	0	0	0
drawTableHeader	0	0	0	0	1	0	0	0	0
drawTableCell	0	0	0	0	0	0	0	0	0
drawTableLine	0	0	0	0	0	0	0	0	0
drawTableLines	0	0	0	1	1	0	0	0	0
drawFactureTail	0	0	0	1	1	0	0	0	0
drawFactureTable	0	0	0	0	0	0	0	0	0
drawFootnote	0	0	0	0	1	0	0	0	0
drawFacturePage	0	0	0	0	0	0	0	0	0
Print	0	0	1	1	1	0	0	0	0
getPageCount	0	0	0	0	1	0	0	0	0
PreviewPanel									
setPageNum	0	0	0	0	0	1	0	0	0
getPageNum	0	0	0	0	0	1	0	0	0
getPageCount	0	0	0	0	0	0	1	0	0
Printing									
doPrinting	0	0	0	0	0	0	0	0	0
createA4Paper	0	0	0	0	0	0	0	1	0
FacturesDebiteursPrinting									
getStrValue	0	0	0	0	0	0	0	0	0
drawTitle	0	0	0	0	0	0	0	0	2
drawEtablissement	0	0	0	0	0	0	0	0	4
drawDestinataire	0	0	1	0	0	0	0	0	2
drawFactureInfo	0	0	2	0	0	0	0	0	3
drawFactureHeading	0	0	0	0	0	0	0	0	0
drawOneLineCell	0	0	0	0	0	0	0	0	0
drawTableHeader	0	0	0	0	0	0	0	0	1
drawTableCell	0	0	0	0	0	0	0	0	0
drawTableLine	0	0	0	0	0	0	0	0	0
drawTableLines	0	0	0	1	0	0	0	0	2
drawFactureTail	0	0	0	1	0	0	0	0	3
drawFactureTable	0	0	0	0	0	0	0	0	1

drawTalon	0	0	1	0	0	0	0	0	3
drawFootnote	0	0	0	0	0	0	0	0	1
DrawFacturePage	0	0	0	0	0	0	0	0	0
Print	0	0	1	1	0	0	0	0	1
getPageCount	0	0	0	1	0	0	0	0	1

Table 6.6: Similarity matrix for all classes and methods inside *Environment* package.

We applied our approach on the previous similarity matrix. The approach suggested only one update, which is to move the method `drawFactureTableLines` from class `FacturesEmisesPrinting` to class `FilteredDataProvider`. Although this will increase cohesion and decrease coupling, still it is not the expected output from this approach. However, this reflects the fact that many methods not use attributes from the classes. This is notable in the previous table since it has many zero lines. In the previous table there are 24 zero lines. This is means that these methods use no attribute from the classes. Thus, they will not give us many improvements.

6.4.2. Experimental Results on Software Refactoring at Class Level Using Clustering Without Fixed Cluster-Center

In this section we present the results of our proposed approach. The testing was done through two steps. In the first step, we used traditional clustering techniques as an engine of the clustering phase. The clustering techniques which were used are SLINK, CLINK, and WPGMA. We conducted three experiments, each using one of those clustering techniques. In the second step, we used our proposed algorithm A-KNN as an engine for the clustering phase. A-KKN showed a better performance than SLINK, CLINK, WPGMA when it was applied at the function level. Hence, we conducted a set of

experiments to test the efficiency of A-KNN at the class level. The results were compared with the performance of all algorithms to determine the best engine for the clustering phase. We tested the technique on an industrial system, called Java Line Of Code (JLOC). JLOC is made in Java, and it provides analysis of the Lines Of Code (LOC) for any project. Currently, C++, Java, VB, SQL, Makefile and Matlab files are supported. It will count the total number of comment lines, blank lines and actual source code lines. It is available in [63]. The system consists of five classes `BasicFileInfo`, `CommonCounter`, `Gui`, `class`, `Table` and one interface `ILineCounter`. The classes contain 27 methods and 23 data-members. Table 6.7 shows the entity-attribute matrix for the system. Table 6.8 shows the similarity matrix for the system. The table consists of 27 rows and 27 columns, and it represents the similarity between 27 distinct entities.

		Fcomments	Fblanks	Fsourcelines	fname	ftotal	Inputfile	Frame	Panel	panel1	fchooser	Label	Browse	go	Fname	filename	al	m_colNames	m_data	num_rows	num_cols	filename	guiobj	list
1	getFcomments	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	setFcomments	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	getFblanks	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	setFblanks	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	getFsourcelines	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	setFsourcelines	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	getFname	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	setFname	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	getFtotal	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	setFtotal	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	CommonCounter	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	Countlines	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	jbInit	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	initComponents	0	0	0	0	0	0	8	7	3	0	2	3	3	3	0	0	0	0	0	0	0	0	0
15	Main	0	0	0	0	0	0	8	7	3	0	2	3	3	3	0	0	0	0	0	0	0	0	0
16	actionPerformed	0	0	0	0	0	0	3	0	0	6	0	1	3	3	0	0	0	0	0	0	0	0	0
17	Main	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	1	0	0	0	0	0	0	0
18	FileIterator	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0
19	Table	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
20	getRowCount	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
21	getColumnCount	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
22	getValueAt	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0
23	setData	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0
24	setColumnName	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
25	getColumnName	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
26	Run	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	1	8	9	10	10	1	2	3
27	Show	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	1	8	9	10	10	1	2	3

Table 6.7: Entity-Attribute Matrix of the system JLOC [63]

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
3	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
4	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
5	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
6	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
7	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
8	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
9	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
11	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
12	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
13	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
14	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.5	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.87	0.87
15	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.5	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.87	0.87
16	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.5	0.5	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.85	0.85
17	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.86	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
18	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.86	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.92	0.92
19	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.86	0.86
20	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.88	0.88
21	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.88	0.88
22	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.62	0.62
23	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.62	0.62
24	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.88	0.88
25	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.88	0.88
26	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.87	0.87	0.85	1.0	0.92	0.86	0.88	0.88	0.62	0.62	0.88	0.88	1.0	1.0
27	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.87	0.87	0.85	1.0	0.92	0.86	0.88	0.88	0.62	0.62	0.88	0.88	1.0	1.0

Table 6.8: Similarity matrix of the system JLOC [63]

6.4.2.1. Experiments of the approach using SLINK, CLINK, WPGMA

In this section we present the results of our investigation into the efficiency of the proposed approach using SLINK, CLINK, WPGMA algorithms in the clustering phase. Figure 6.6, Figure 6.7 and Figure 6.8 show the clustering tree for the three algorithms SLINK, CLINK, WPGMA algorithms respectively.

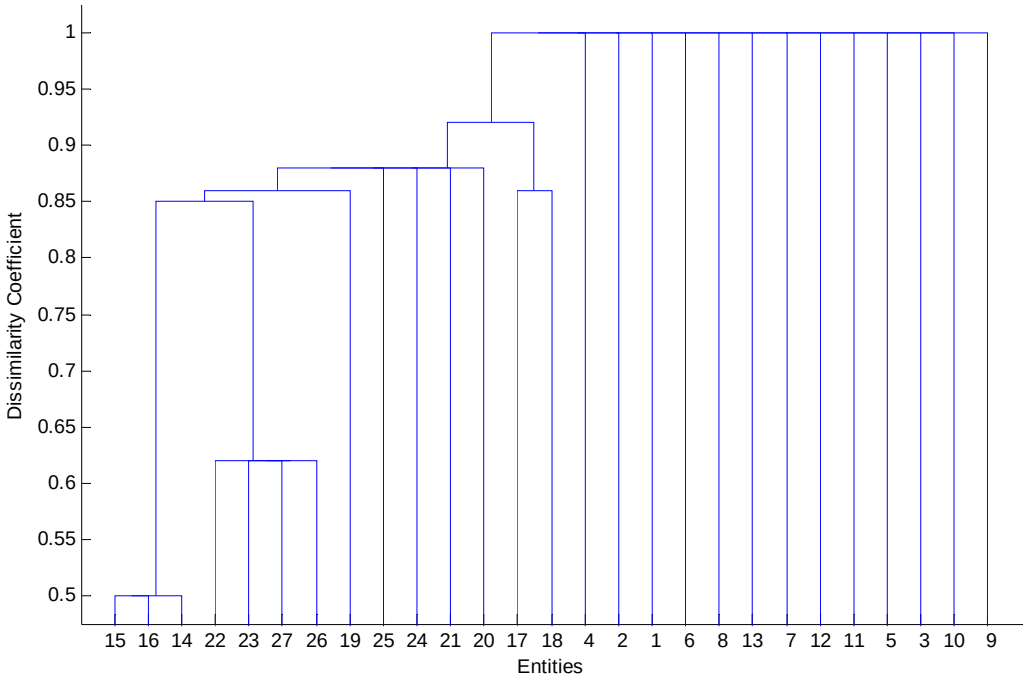


Figure 6.6: Clustering Hierarchy of the system JLOC [63] using SLINK algorithm

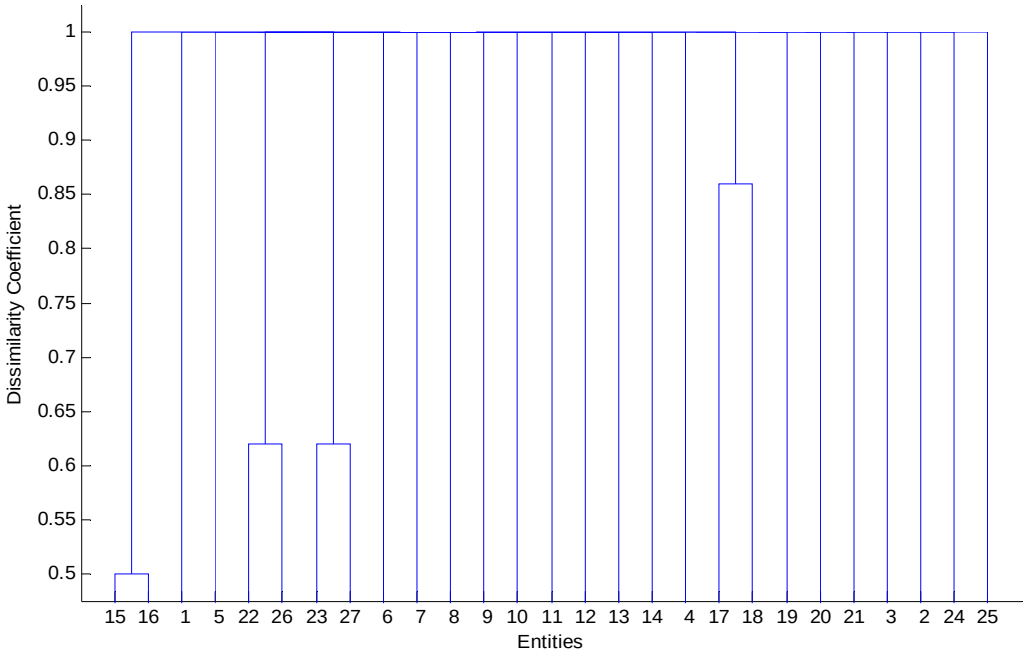


Figure 6.7: Clustering Hierarchy of the system JLOC [63] using CLINK algorithm

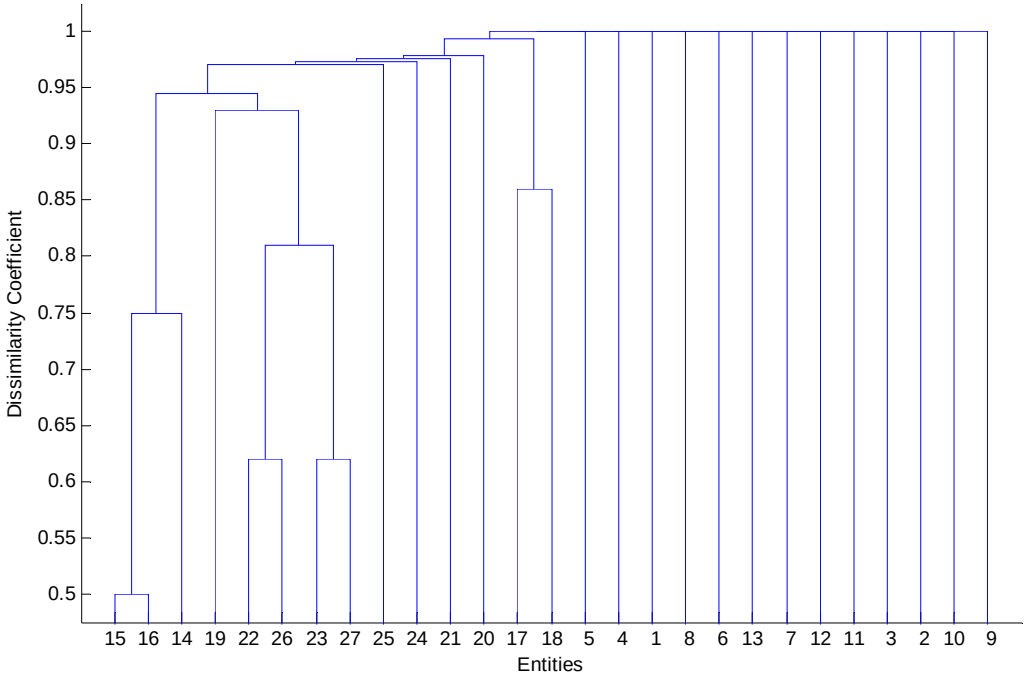


Figure 6.8: Clustering Hierarchy of the system JLOC [63] using WPGMA algorithm

SLINK and WPGMA showed excellent behavior. Both of them grouped methods 14, 15 and 16 in one cluster, methods 22, 23, 26 and 27 in another cluster, and methods 17 and 18 in a third cluster. The performance of CLINK was less good than SLINK and WPGMA. CLINK merged clusters 22 and 26 into one cluster, and clusters 23 and 27 into another. However, it did not merge the two clusters into the same cluster. Moreover, CLINK did not merge clusters 14, 15 and 16 correctly. However, SLINK clustering was easier to read. This saves the time required by the software designer to conclude how refactoring can be done. Thus, SLINK has an advantage over WPGMA.

6.4.2.2. Experiments of the approach using A-KNN

In chapter 5, we tested the performance of A-KNN on an industrial system. We compared its performance with the performance of SLINK, CLINK and WPGMA. The introduced algorithm A-KNN showed high competitive performance compared with WPGMA. In addition, it has less computational complexity than other algorithms. The advantages of the implemented A-KNN over the other clustering algorithms led us to investigate the performance of A-KNN as an engine for the clustering phase in our new presented approach for software refactoring at the class level using a variable number of classes. Figure 6.9 shows the output of A-KNN.

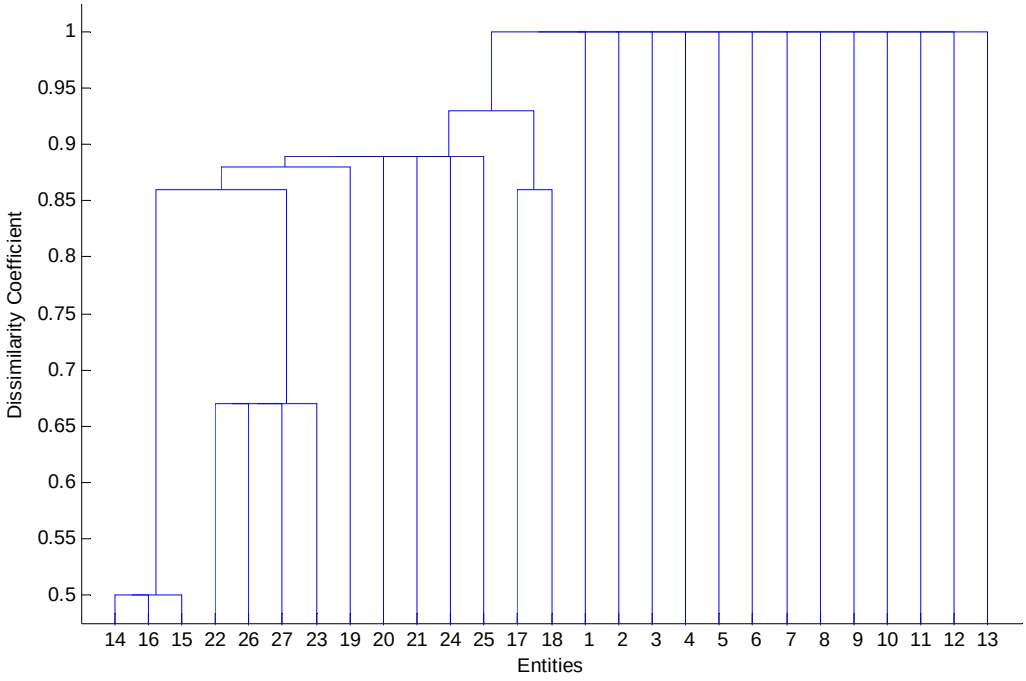


Figure 6.9: Clustering Hierarchy of the system JLOC [63] using A-KNN algorithm

A-KNN showed very similar output to SLINK. It grouped methods 14, 15 and 16 into one cluster, methods 22, 23, 26 and 27 into another, and methods 17 and 18 into a third cluster. Even the clusters 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13 were kept in the same cluster. Thus, A-KNN is better than the three traditional SLINK, CLINK and WPGMA clustering algorithms. Moreover, as was mentioned in chapter 5, A-KNN requires less computation.

6.4.2.3. Analysis of the results

We applied refactoring based on the results of A-KNN and SLINK, since they showed the best performance and their results are very similar. Appendix A shows the source code before and after refactoring. A new class was extracted, called *TableSchema*. This class

contains methods 12, 20, 24 and 25 (*getRowCount, getColumnCount, setColumnName, getColumnName*). We measured the values of coupling before and after refactoring. We used Coupling Between Object Classes (CBO) as a measure of coupling. CBO is a count of the number of other classes to which a class is coupled. It is measured by counting the number of distinct non-inheritance related class hierarchies on which a class depends. We used CBO instead of CTA because there was no coupling through abstraction between the classes. Table 6.9 shows the CBO and LCOM values before and after refactoring.

	Before Refactoring		After Refactoring	
Node	CBO	LCOM	CBO	LCOM
BasicFileInfo.java	0	45	0	45
CommonCounter.java	2	4	2	4
Gui.java	1	4	1	4
ILineCounter.java	1	0	1	0
Main.java	2	0	2	0
Table.java	3	0	4	0
TableSchema.java			0	0

Table 6.9: CBO and LCOM values of the system JLOC [63] before and after refactoring

Table 6.9 shows that there is an increase in the values of CBO but the values of LCOM are still the same. However, this increase in coupling is normal, since we have a new class added to the system, which is *TableSchema*. The class *Table* is coupled by the new class *TableSchema*. Thus, there is an increase by 1 in the CBO of the class *Table*. The

technique is useful in enhancing the understandability of the code. This is because the use of two classes `Table` and `Tableschema,` instead of one class `Table,` decreases the maintenances effort due to the decrease of code complexity.

6.5. Conclusions

In this chapter, we investigated software refactoring at the class level using pattern recognition techniques. We proposed two new approaches to guide the software designer in refactoring at the class level. The first approach is "software refactoring at class level with fixed number of classes" and the second approach is "software refactoring at class level using variable number of classes". We explored the different settings and controllers of these approaches. We conducted a set of experiments to check the efficiency of these approaches. The experiments were conducted on different experimental units from different sources. The approaches were proven to be helpful in both correcting the errors in the current system structure and providing a new design for the system classes. Our proposed algorithm A-KNN continued to show its superiority over the traditional SLINK, CLINK, and WPGMA clustering algorithms.

CHAPTER 7

SOFTWARE REFACTORING AT THE PACKAGE LEVEL

7.1. Introduction

In the previous chapter, we applied the clustering technique at the class level. The entities to be clustered were the classes' methods. In this chapter, we present our investigation of software refactoring by using the clustering technique at a higher level which is the architecture level. Two approaches are presented that aid in software refactoring at the package level. The goal is to achieve architectural design in which the packages have a high level of intra-package cohesion and a low level of inter-package coupling. Thus, the aim of the presented approaches is to make a balance between the cohesion inside the package and the coupling between packages.

Our first approach is called Software Refactoring at the Package Level Using Clustering with Fixed Number of Packages. It aims to redistribute the classes on the packages by adapting the clustering technique so that it balances coupling and cohesion. Thus, the

number of classes inside each package will change, because some classes will be moved from their original packages to other packages inside the system. Hence, the number of clusters equals the number of packages. Consequently, the entities to be clustered are the classes, and the cluster centers represent the packages. The cluster (class) will be assigned to a cluster center (package) according to its similarity with that package. The similarity between a class and a package is the number of the instances of that class used as classes attributes inside the package.

The second approach is called Software Refactoring at the Package Level Using Variable Number of Packages. The classes will be clustered according to their similarity with each other. The approach assigns similar classes to the same group. At the end, the approach provides a set of packages (clusters) and, inside each of them, there is a set of similar classes. Each of these clusters will be mapped to a complete package during refactoring.

The rest of this chapter is organized as follows. Section 2 describes our first approach (Software Refactoring at the Package Level Using Clustering with Fixed Number of Packages). Section 3 explains the second approach (Software Refactoring at the Package Level Using Clustering with Variable Number of Packages). Section 4 explains the experimental results. Section 5 provides the conclusions.

7.2. Software Refactoring at the Package Level Using Clustering with Fixed Number of Packages

In this approach, the entities to be clustered are the classes inside the packages. One class inside a package may communicate with any other class inside the package with different types of communication (e.g. inheritance, association, aggregation). However, our concerns are about the relations in which a class uses an instance of other class (inside the same package or in other package) as one of its attributes. Indeed, relations between classes inside the package represent the coupling between classes if we are looking at the class level. If we are looking at package level, then this type of relations will represent the cohesion inside the package. The relations to classes inside other packages will represent the coupling. Thus, the more relations between classes inside the package, the more cohesive is the package. The more relations between the package and other packages, the more coupled is the package. The approach depends mainly on extracting the similarity matrix for the source code entities and attributes.

The entities to be clustered are the classes. The entities will be clustered according to their similarity with the packages. The class is similar to a package if many instances of that class are used as attributes by other classes of the package. Thus, we have a fixed number of cluster centers, which is the number of current packages in the system, and all entities (classes) should be assigned to these clusters. Consequently, the package represents the cluster center. For example, if we have three packages, each containing five classes, and if we want to apply our approach on those packages, then we will have three cluster centers

and 15 entities to be assigned to these centers. Each entity will be assigned to a cluster center according to its similarity with that center. For instance, if the number of instances of class A inside package P1 is larger than the number of instances of that class in package P2, then class A will be more similar to P1 than to P2.

The attributes are the packages to which we want to distribute the classes. Thus, in the similarity matrix, the rows represent the classes and the columns represent the packages. If the value in the similarity matrix is 0, then the class does not satisfy the attribute. In other words, no instances of this class are used as data members for the classes of that package. On the other hand, if the value inside the similarity matrix is non-zero, then the class satisfies the attributes. This means that there are classes inside the package which use instances of this class as data members.

The algorithm of the approach can be summarized in the following steps:

- Calculate the similarity matrix between the classes and the packages.
- If the maximum similarity for the class is with the containing package
    - Keep the class inside the package
- Otherwise
    - Assign each class to the package which is most similar to it.

The similarity between a class and package is given by the following formula:

$$Sim(A, P)= w$$

where:

A: is the class

P: is the package

w: is number of instances of class A used as attributes by the classes in the package P

7.3. Software Refactoring at the Package Level Using Clustering With Variable Number of Packages

In the previous sections, we presented an approach to perform clustering at the package level. The presented approach redistributed the classes over the original system packages, and thus it kept the same number of the packages in the system. In the following sections, we will present an approach for "auto-packaging" by suggesting a new arrangement of the packages and classes. This auto-packaging for the classes is guided by inter-package cohesion and intra-package coupling. The clustering technique suggests different designs with different package diagrams. This enables the software designer to choose a new design for the current system.

7.3.1. The approach

This section provides an approach for package refactoring by using pattern recognition techniques, and it explains the different aspects of this approach.

7.3.1.1. Package Refactoring Approach

This approach provides an architectural design of the system as a hierarchical tree. It approach provides information about the existing packages of the system and heuristic guide-lines to improve the current architectural design. These guidelines can be used to help the software designer in how to refactor the current system. The main elements of the system are the entities and their attributes. The entities are the components to be clustered, and the attributes are the properties of these entities. After extracting the entities and their attributes, a set of matches between these entities determines the similarity between them. Based on this similarity, the clustering technique groups the different entities within clusters. We used SLINK, CLINK and WPGMA clustering techniques. Then, we used our proposed algorithm A-KNN.

7.3.1.2. Entities

For software refactoring at the package level, classes were chosen as entities. Classes are the basic entities of system, and the goal is to find the best arrangement to package these classes in such a way as to increase the intra-package cohesion and to reduce the inter-package coupling.

7.3.1.3. Attributes

To perform clustering at the package level, the possible attributes for the entities should be identified. Different class methods may be related to different functional tasks. Therefore, the classes' methods are used as attributes for the entities. An entity has an access to all methods inside the class. It can also access methods of other classes by using

instances from those classes. In both cases, we are interested in how the classes use the methods from either the same class or other classes. Based on this, the attribute is a method attached with the label of the containing class. Each attribute is measured as a quantitative scale representation. The attribute value is the number of times the class accesses this method represented by an attribute.

$$V(att_i, ent_j) = A(att_i, ent_j);\ i \in [1..a], j \in [1..b]$$

where:

$V(att_i, ent_j)$: is the value of the attribute att_i for the entity ent_j

a: is the number of the methods in the system

b: is the number of the classes in the system

$A(att_i, ent_j)$: is the number of accesses to the method represented by the attribute att_i in the class represented by the entity ent_j

Based on the previous discussion, the entities-attributes matrix will be filled. Hence, between any two entities in the entities-attribute matrix, there are three types of matches:

0-n/n-0 match: the attribute is present n times in one entity and absent from the other entity (n>0)

n-m match: the attribute is present n times in one entity and m times in the other entity (n>0, m>0).

0-0 match: the attribute is absent from the two entities.

7.3.1.4. Similarity Measure

For a system consisting of a set of packages, there is only one entity-attribute matrix for the whole system. The columns represent all the attributes (classes' methods) in the

system. The rows represent all classes in the packages. To calculate the similarity between the rows of the entity-attribute matrix, we used the same similarity measure which was presented in Chapter 6.

7.4. Experimental Results

This section includes our experimental results on the two approaches. The first approach was tested in two steps. The first step was on a made-up source code, and the second step on real industrial source code. The second approach was tested immediately on the industrial system.

7.4.1. Experimental Results on Software Refactoring at the Package Level Using Clustering with Fixed Number of Packages

In this work, we applied our technique to a made-up source code to present the used technique. Then, we applied the technique to the industrial system. The following subsections present our work.

7.4.1.1. Experimental Results on Test Source Code

We applied our approach on a test source code. Appendix A contains the source code before refactoring, and appendix B shows the source code after refactoring. The source code consists of 3 packages (P1, P2, P3) and each package contains 5 classes. Table 7.1 shows the classes in each package.

Package	Classes
P1	A, B, C, D, E
P2	F,G, H, I, J
P3	K, L, M, N, O

Table 7.1: Packages and classes in each package of the test source code

We calculated the similarity matrix between each class and the three packages. Table 7.2 shows the similarity matrix.

	P1	P2	P3
A	1	1	0
B	0	0	0
C	0	0	0
D	0	0	3
E	0	0	0
F	5	2	0
G	0	1	3
H	0	0	0
I	0	2	0
J	0	1	0
K	1	3	1
L	1	0	0
M	0	0	0
N	0	0	1
O	0	0	2

Table 7.2: Similarity matrix of the test source code

We applied our clustering approach. The class will be included in the package which is most similar according to the similarity values in the table. For example, in the previous table, class A is most similar to package P1, and so we will keep it in package P1. Class F is most similar to package P1, and so it is moved from package P2 to P1. Table 7.3 shows the new distribution of the classes on the packages.

Package	Classes
P1	A, B, C, E, F, L
P2	H, I, J, K
P3	D, G, M, N, O

Table 7.3: New distribution of the classes on the packages of the test source code

Table 7.4 shows the number of connections among classes inside the package and the number of connections to other packages before refactoring.

	P1	P2	P3
No of connections inside the package	1	6	4
No of connections to other packages	7	4	6

Table 7.4: Connections among classes of the test source code before refactoring

Table 7.5 shows the number of connections among classes inside the package and the number of connections to other packages after refactoring.

	P1	P2	P3
No of connections inside the package	7	6	9
No of connections to other packages	1	4	1

Table 7.5: Connections among classes of the test source code after refactoring

Table 7.6 shows the change in the number of classes as the results of refactoring.

	P1	P2	P3
No of connections inside the package	+ 6	0	+5
No of connections to other packages	-6	0	-5

Table 7.6: Change in the connections number of the test source code after refactoring

From the previous table, we can conclude that the number of connections inside the packages has increased and the number of connections to other packages has decreased. Hence, we can say that the technique increased the cohesion inside the package and decreased the coupling between packages.

7.4.1.2. Experimental Results using Industrial System

We applied our approach on an open source system (viz. *Front End for My SQL Domain1.0*). The system *Front End for MYSQL* is the portable front end to the open source database server MYSQL. It aims to provide a complete front end to MYSQL, and thus it reduces the difficulty of interacting with the system using SQL query language. This system is available at [64], and it consists of 9 packages and 53 classes. The packages are *BackEnd, System, DataStructures, DriverModule, Editor, IO, XMLutil, BackEndData, BackEndInterfaces.* Table 7.7 shows the similarity matrix between the classes and the packages of the system. The columns represent the packages, and the rows represent the classes.

Class / Package	BackEnd	System	DataStructures	DriverModule	Editor	IO	XMLutil	BackEnd Data	BackEnd Interfaces
BackEnd									
BackEnd	1	0	0	0	0	0	0	0	0
Database Reader	0	0	0	0	0	0	0	0	0
JDBC2_ ConnectionManager	1	0	0	0	0	0	0	0	0
JDBC_Connection Manager	0	0	0	0	0	0	0	0	0
ProcedureCompiler	0	0	0	0	0	0	0	0	0
ProcedureExecuter	0	0	0	0	0	0	0	0	0
ProcedureGenerator	0	0	0	0	0	0	0	0	0
QueryExecuter	1	0	0	0	0	0	0	0	0
QueryGenerator	2	0	0	0	0	0	0	0	0
TransactionManager	1	0	0	0	0	0	0	0	0
UserManager	1	0	0	0	0	0	0	0	0
System									
InitialDriverInfo	0	0	0	0	0	0	0	0	0

SystemInformationDatabase	0	0	0	0	0	0	0	0	0
SystemInformationProvider	0	0	0	0	0	0	0	0	0
SystemInitializer	0	0	0	0	0	0	0	0	0
BackEndComponent									
DataStructures									
StatusBarDataStructure	0	0	0	0	0	0	0	0	0
DriverModule									
DriverManager	0	0	0	0	0	0	0	0	0
DriverManagerInterface	1	0	0	0	0	0	0	0	0
DriverXMLDatabase	0	0	0	0	0	0	0	0	0
InvalidJAR	0	0	0	0	0	0	0	0	0
JarClassLoader	0	0	0	1	0	0	0	0	0
JDBCDriverLoader	0	0	0	0	0	0	0	0	0
Editor									
DocumentEditor	0	0	0	0	0	0	0	0	0
EditorFormator	0	0	0	0	0	0	0	0	0
TreeSearcher	0	0	0	0	0	0	0	0	0
IO									
IOManager	1	0	0	0	0	0	0	0	0
IOUtil	0	0	0	0	0	0	0	0	0
XMLutil									
XMLWriter	0	0	0	1	0	0	0	0	0
BackEndData									
Database	0	0	0	0	0	0	0	0	0
Driver	0	0	0	0	0	0	0	0	0
Foreign_key	0	0	0	0	0	0	0	0	0
Procedure	0	0	0	0	0	0	0	0	0
Table	0	0	0	0	0	0	0	1	2
TableAttribute	0	0	0	0	0	0	0	1	0
TableTrigger	0	0	0	0	0	0	0	0	0
Transaction	1	0	0	0	0	0	0	0	0
Tuple	0	0	0	0	0	0	0	0	3
User	0	0	0	0	0	0	0	0	0
View	0	0	0	0	0	0	0	1	1
BackEndInterfaces									
DatabaseInterface	0	0	0	0	0	0	0	0	0
DatabaseTableCha	0	0	0	0	0	0	0	0	0

ngeInterface									
DatabaseViewChangeInterface	0	0	0	0	0	0	0	0	0
DriverListener	0	0	0	0	0	0	0	0	0
ForeignKeyChangeInterface	0	0	0	0	0	0	0	0	0
ForeignKeyInterface	0	0	0	0	0	0	0	1	1
ProcedureInterface	0	0	0	0	0	0	0	0	0
TableAttributeChangedInterface	0	0	0	0	0	0	0	0	0
TableAttributeInterface	0	0	0	0	0	0	0	1	2
TableInterface	0	0	0	0	0	0	0	0	0
TableTriggerInterface	0	0	0	0	0	0	0	1	2
TableValueChange	0	0	0	0	0	0	0	0	0
TransactionListener	1	0	0	0	0	0	0	0	0
TriggerChangeInterface	0	0	0	0	0	0	0	0	0

Table 7.7: Similarity Matrix for the classes and packages in the system Front End For My SQL Domain1.0 [64] before refactoring

Applying our approach suggested a set of actions, shown in Table 7.8.

No	Class	Old Package	New Package
1	IOManager	IO	BackEnd
2	DriverManagerInterface	DriverModule	BackEnd
3	XMLWritter	XMLutil	DriverModule
4	Table	BackEndData	BackEndInterface
5	TableAttribute	BackEndData	BackEndInterface
6	Tuple	BackEndData	BackEndInterface

Table 7.8: Actions suggested by applying the approach

Table 7.9 shows the similarity matrix of the system after refactoring. As a result of refactoring, one of the packages **(***XMLutil)* was removed from the system.

Class / Package	BackEnd	System	DataStructures	DriverModule	Editor	IO	BackEnd Data	BackEnd Interfaces
BackEnd								
BackEnd	1	0	0	0	0	0	0	0
DatabaseReader	0	0	0	0	0	0	0	0
JDBC2_ConnectionManager	1	0	0	0	0	0	0	0
JDBC_ConnectionManager	0	0	0	0	0	0	0	0
ProcedureCompiler	0	0	0	0	0	0	0	0
ProcedureExecuter	0	0	0	0	0	0	0	0
ProcedureGenerator	0	0	0	0	0	0	0	0
QueryExecuter	1	0	0	0	0	0	0	0
QueryGenerator	2	0	0	0	0	0	0	0
TransactionManager	1	0	0	0	0	0	0	0
UserManager	1	0	0	0	0	0	0	0
IOManager	1	0	0	0	0	0	0	0
DriverManagerInterface	1	0	0	0	0	0	0	0
System								
InitialDriverInfo	0	0	0	0	0	0	0	0
SystemInformationDatabase	0	0	0	0	0	0	0	0
SystemInformationProvider	0	0	0	0	0	0	0	0
SystemInitializer	0	0	0	0	0	0	0	0
DataStructures								
StatusBarDataStructure	0	0	0	0	0	0	0	0
DriverModule								
DriverManager	0	0	0	0	0	0	0	0
DriverXMLDatabase	0	0	0	0	0	0	0	0
InvalidJAR	0	0	0	0	0	0	0	0
JarClassLoader	0	0	0	1	0	0	0	0
JDBCDriverLoader	0	0	0	0	0	0	0	0

XMLWriter	0	0	0	1	0	0	0	0
Editor								
DocumentEditor	0	0	0	0	0	0	0	0
EditorFormator	0	0	0	0	0	0	0	0
TreeSearcher	0	0	0	0	0	0	0	0
IO								
IOUtil	0	0	0	0	0	0	0	0
BackEndData								
Database	0	0	0	0	0	0	0	0
Driver	0	0	0	0	0	0	0	0
Foreign_key	0	0	0	0	0	0	0	0
Procedure	0	0	0	0	0	0	0	0
TableAttribute	0	0	0	0	0	0	1	0
TableTrigger	0	0	0	0	0	0	0	0
Transaction	1	0	0	0	0	0	0	0
User	0	0	0	0	0	0	0	0
View	0	0	0	0	0	0	1	1
BackEndInterfaces								
DatabaseInterface	0	0	0	0	0	0	0	0
DatabaseTableChangeInterface	0	0	0	0	0	0	0	0
DatabaseViewChangeInterface	0	0	0	0	0	0	0	0
DriverListener	0	0	0	0	0	0	0	0
ForeignKeyChangeInterface	0	0	0	0	0	0	0	0
ForeignKeyInterface	0	0	0	0	0	0	1	1
ProcedureInterface	0	0	0	0	0	0	0	0
TableAttributeChangedInterface	0	0	0	0	0	0	0	0
TableAttributeInterface	0	0	0	0	0	0	1	2
TableInterface	0	0	0	0	0	0	0	0
TableTriggerInterface	0	0	0	0	0	0	1	2
TableValueChange	0	0	0	0	0	0	0	0

TransactionListener	1	0	0	0	0	0	0	0
TriggerChangeInterface	0	0	0	0	0	0	0	0
Table	0	0	0	0	0	0	1	2
Tuple	0	0	0	0	0	0	0	3

Table 7.9: Similarity Matrix for the classes and packages in the system Front End For My SQL Domain1.0 [64] after refactoring

Table 7.10 shows the number of connections among classes inside the package and the number of connections to other packages before refactoring.

	BackEnd	System	DataStructures	DriverModule	Editor	IO	XMLutil	BackEnd Data	BackEnd Interfaces
No of connections inside the package	7	0	0	1	0	0	0	3	5
No of connections to other packages	4	0	0	1	0	0	0	3	6

Table 7.10: Number of connections of the system Front End for My SQL Domain1.0 [64] before refactoring

Table 7.11 shows the number of connections among classes inside the package and the number of connections to other packages after refactoring.

	BackEnd	System	DataStructures	DriverModule	Editor	IO	BackEnd Data	BackEnd Interfaces
No of connections inside the package	9	0	0	2	0	0	2	10
No of connections to other packages	2	0	0	0	0	0	4	1

Table 7.11: Number of connections of the system Front End for My SQL Domain1.0 [64] after refactoring

Table 7.12 shows the change in the number of classes as the results of refactoring.

	BackEnd	System	DataStructures	DriverModule	Editor	IO	BackEnd Data	BackEnd Interfaces
No of connections inside the package	+2	0	0	+1	0	0	-1	+5
No of connections to other packages	-2	0	0	-1	0	0	+1	-5

Table 7.12: Change in the connections number of the system Front End for My SQL Domain1.0 [64] after refactoring

From the previous table, we can notice that our approach enhanced the cohesion inside the package by increasing the number of connections inside the package. In addition, it decreased the coupling between packages by decreasing the number of connections to other packages.

7.4.2. Experimental Results on Software Refactoring at the Package Level Using Clustering with Variable Number of Packages

Using the same testing methodology as in chapter 6, we tested the approach in two steps. The first step used the SLINK, CLINK and WPGMA clustering algorithms, and the second step used our implemented algorithm A-KNN. We tested the approach on the open source system called Trama. Trama is a tool to work with matrixes. It also provides different graphical user interfaces to work graphically with matrixes. It is available on [65]. Trama consists of six packages and fifteen classes distributed between these packages. Table 7.13 shows the packages and the classes inside each package in the original source code. Each class was given a unique identity to be used in the clustering technique.

Package ID	Package	Class ID	Class
1	Negocio	1	ControleProjeto
		2	ControleTela
		3	Main
		4	Matriz
2	negocio.leitor	5	LeitorDeModelo
3	negocio.leitor.Interface	6	PluginInterface
4	Persistencia	7	DadosMatriz
		8	PersistenciaProjeto
		9	Projeto
5	Visao	10	JTableCustomizado
		11	ModeloTabela
		12	Tela
6	visao.renderizador	13	RenderizadorCelula
		14	RenderizadorTituloColuna
		15	RenderizadorTituloLinha

Table 7.13: Packages and classes inside each package of the system Trama [65]

Table 7.14 shows the number of connections among classes inside each package and the number of connections to other packages in the original source code. A connection inside a package is an instance of a class belonging to the same package. The connection to other package is an instance of a class belonging to another package.

	negocio	negocio.leitor	negocio.leitor.Interface	Persistencia	Visao	visao.renderizador	Total Packages
No of connections inside the package	4	0	0	0	9	0	13
No of connections to other packages	7	0	0	0	4	0	11

Table 7.14: Number of connections in the original source code of the system Trama [65]

The total number of methods in the classes is two hundred. Thus, the entity-attribute matrix contains 15 rows and 200 columns. After extracting the entity-attribute matrix, we

calculated the similarity and dissimilarity between the different entities. Table 7.15 shows the dissimilarity matrix for the system. Since we have 15 different classes in the system, the table consists of 15 rows and 15 columns, and it represents the dissimilarity between 15 distinct entities, each representing one class.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1.0	0.51	1.0	0.37	1.0	1.0	0.81	0.99	0.99	0.95	0.86	0.53	1.0	1.0	1.0
2	0.51	1.0	1.0	0.87	0.98	0.98	1.0	1.0	1.0	1.0	0.98	0.17	0.97	0.97	0.97
3	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
4	0.37	0.87	1.0	1.0	1.0	1.0	0.62	1.0	1.0	0.96	0.86	0.69	1.0	1.0	1.0
5	1.0	0.98	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
6	1.0	0.98	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.9	1.0	0.98	1.0	1.0	1.0
7	0.81	1.0	1.0	0.62	1.0	1.0	1.0	1.0	1.0	1.0	0.82	0.93	1.0	1.0	1.0
8	0.99	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.89	1.0	0.98	1.0	1.0	1.0
9	0.99	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.9	1.0	0.98	1.0	1.0	1.0
10	0.95	1.0	1.0	0.96	1.0	0.9	1.0	0.89	0.9	1.0	0.93	0.82	0.86	0.83	0.83
11	0.86	0.98	1.0	0.86	1.0	1.0	0.82	1.0	1.0	0.93	1.0	0.97	1.0	1.0	1.0
12	0.53	0.17	1.0	0.69	1.0	0.98	0.93	0.98	0.98	0.82	0.97	1.0	0.94	0.94	0.94
13	1.0	0.97	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.86	1.0	0.94	1.0	1.0	1.0
14	1.0	0.97	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.83	1.0	0.94	1.0	1.0	1.0
15	1.0	0.97	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.83	1.0	0.94	1.0	1.0	1.0

Table 7.15: Dissimilarity matrix of the system Trama [65]

7.4.2.1. Experimental results using SLINK, CLINK, WPGMA

We applied clustering by using the three clustering algorithms SLINK, CLINK, WPGMA. Figure 7.1, Figure 7.2 and Figure 7.3 show the clustering tree for the three algorithms SLINK, CLINK, WPGMA respectively.

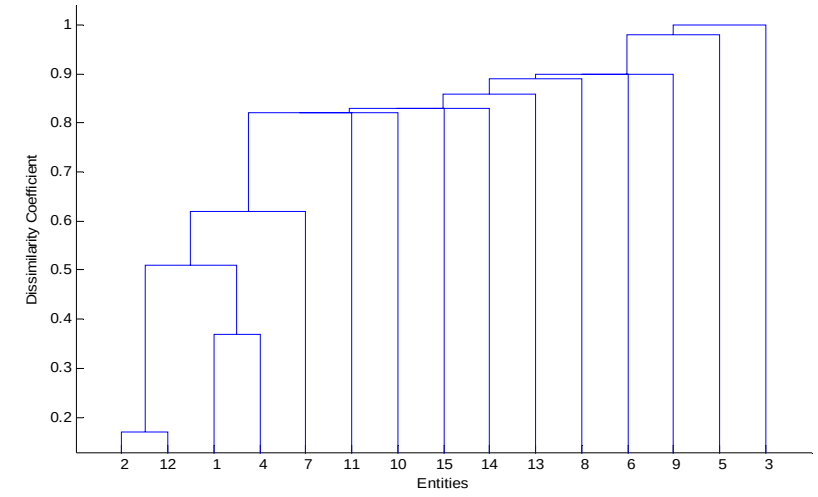


Figure 7.1: Clustering Hierarchy of the system Trama [65] using SLINK algorithm

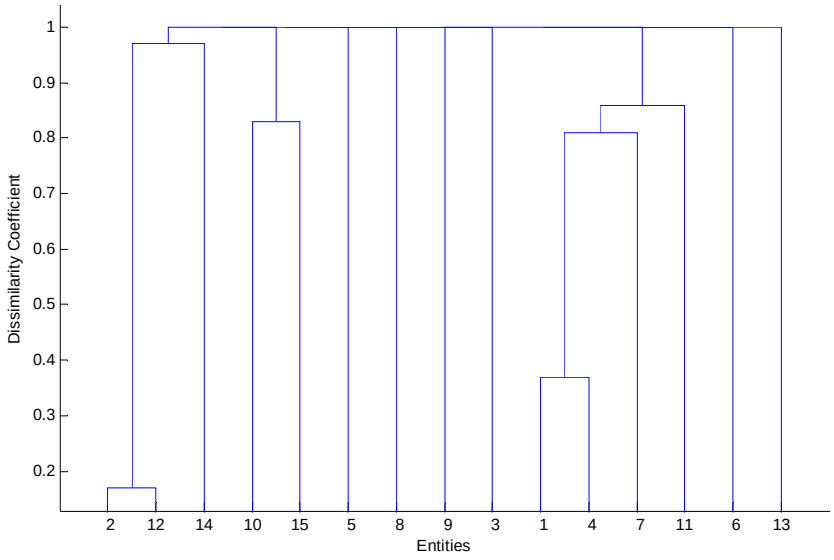


Figure 7.2: Clustering Hierarchy of the system Trama [65] using CLINK algorithm

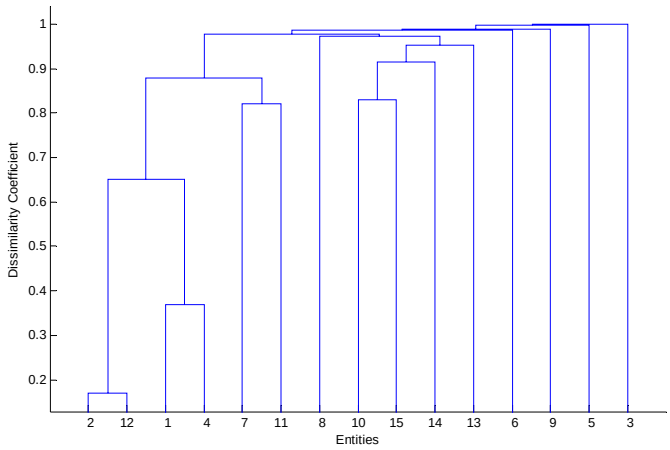


Figure 7.3: Clustering Hierarchy of the system Trama [65] using WPGMA Algorithm

We applied refactoring by using the suggestions provided by the first algorithm SLINK. It suggests many solutions to group the classes. The solutions could be found in the hierarchy between the lowest level and the highest level. At the lowest level, the algorithm put each cluster in a single package. At the highest level, the algorithm put all classes in the same package. We chose the suggested solution at a moderate level. The algorithm suggests 3 packages at a moderate level. Table 7.16 shows the suggested solution.

Package ID	Package	Class ID
1	P1	1, 2, 4, 7,10,11,12
2	P2	6, 8, 9, 13, 14, 15
3	P3	3, 5

Table 7.16: Suggested solution by the algorithm SLINK

Table 7.17 shows the number of connections among classes inside the package and the number of connections to other packages after refactoring.

Connection / package	P1	P2	P3	Total
No of connections inside the package	17	0	0	17
No of connections to other packages	6	0	1	7

Table 7.17: Number of connections of the system Trama [65] after refactoring using the SLINK algorithm

In the next step, we applied refactoring using the suggestions provided by the second algorithm CLINK. At a moderate level in the hierarchy, the algorithm suggests four packages. Table 7.18 shows the packages and classes.

Package ID	Package	Class ID
1	P1	2, 12, 14
2	P2	10, 15
3	P3	1, 4, 7, 11
4	P4	3, 5, 6, 8, 9, 13

Table 7.18: Suggested solution by the CLINK algorithm

Table 7.19 shows the number of connections among classes inside the package and the number of connections to other packages after refactoring.

Connection / package	P1	P2	P3	P4	Total
No of connections inside the package	1	1	5	0	7
No of connections to other packages	12	2	2	1	17

Table 7.19: Number of connections of the system Trama [65] after refactoring using the CLINK algorithm

In the last step, we applied refactoring by using the suggestion provided by the third algorithm WPGMA. As with the previous techniques, the algorithm suggests 3 packages

at a moderate level in the hierarchy. Table 7.20 shows the packages and the classes in each package.

Package ID	Package	Class ID
1	P1	1, 2, 4, 7, 11, 12
2	P2	8, 10, 13, 14, 15
3	P3	3, 5, 6, 9

Table 7.20: Suggested solution by the WPGMA algorithm

Table 7.21 shows the number of connections among classes inside the package and the number of connections to other packages after refactoring.

Connection / package	P1	P2	P3	Total
No of connections inside the package	8	3	0	11
No of connections to other packages	12	0	1	13

Table 7.21: Number of connections of the system Trama [65] after refactoring using the WPGMA algorithm

7.4.2.2. Experimental results using A-KNN

Our implemented algorithm A-KNN was previously used twice. In chapter 5, we tested its performance in software refactoring at the function level. In chapter 6, we tested its performance in software refactoring at the class level. In both cases, the algorithm showed an excellent performance. In this section, we present the results of using this algorithm in software refactoring at the package level. Figure 7.4 shows the output of A-KNN.

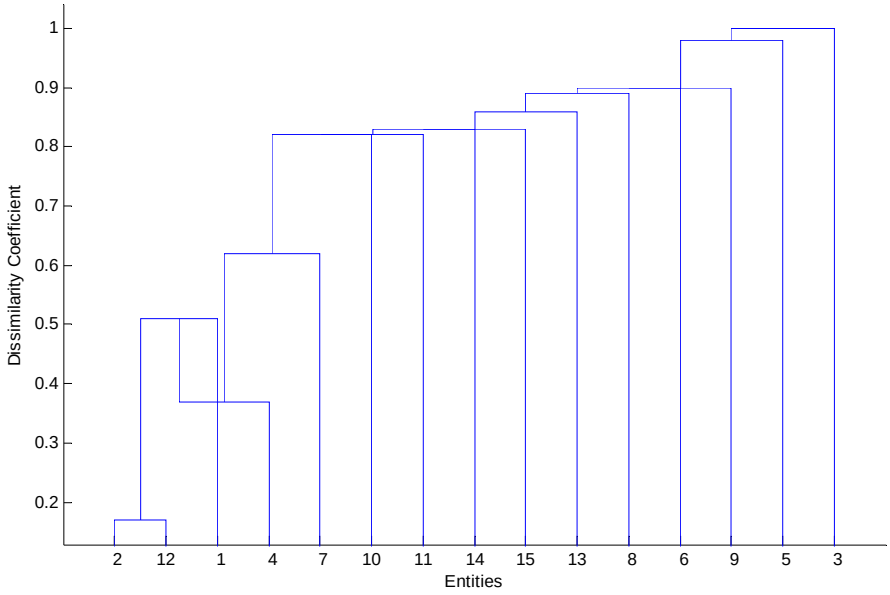


Figure 7.4: Clustering Hierarchy of the system Trama [65] using SLINK algorithm A-KNN

The performance of A-KNN was similar to the performance of SLINK. We applied refactoring by using the suggestions provided by the first algorithm A-KNN. The algorithm suggests many solutions to group the classes. We chose the suggested solution at a moderate level in the hierarchy, where the algorithm suggests 3 packages. Table 7.22 shows the suggested solution.

Package ID	Package	Class ID
1	P1	1, 2, 4, 7, 12
2	P2	6, 8, 9, 10, 11, 13, 14, 15
3	P3	3, 5

Table 7.22: Suggested solution by the A-KNN algorithm

The solution suggested in Table 7.22 was the same as suggested by SLINK. Thus, after refactoring we got the same result as SLINK. Table 7.23 shows the number of

connections among classes inside the package and the number of connections to other packages of the system Trama [65] after refactoring.

Connection / package	P1	P2	P3	Total
No of connections inside the package	8	3	0	11
No of connections to other packages	12	0	1	13

Table 7.23: Number of connections of the system Trama [65] after refactoring using the algorithm A-KNN

7.4.2.3. Analysis of the Results

To summarize the results, Table 7.24 shows the change in the number of connections as results for the refactoring by using the three clustering techniques.

Connection / algorithm	SLINK	CLINK	WPGMA	A-KNN
No of connections inside the package	+4	-6	-2	+4
No of connections to other packages	-4	+6	+2	-4

Table 7.24: Change in the connections number

From the previous table, we can conclude that the proposed approach enhanced the cohesion inside the package by increasing the number of connections inside the package. In addition, it decreased the coupling between packages by decreasing the number of connections to other packages. A-KNN and SLINK were the best two algorithms among the three techniques because they increase the inter-package cohesion by 4 and decrease the intra-package coupling by 4. WPGMA performed better than CLINK, but both WPGMA and CLINK showed poor performance.

7.5. Conclusions

In this chapter, we investigated software refactoring at the package level by using the clustering technique. We proposed two approaches for software refactoring at the package level.

The first approach uses clustering with a fixed number of packages. In this approach, we investigated the similarity measure between the classes and packages. The number of class instances inside a package was proven to be a suitable similarity measure between the class and the package. After proposing the adaptive clustering approach to distribute the entities on the fixed number of cluster centers, we applied our approach on a test code. The results of this proved that the approach is useful in satisfying a balance between coupling and cohesion. Later, on an industrial system, the approach was proved again to be useful in balancing between coupling and cohesion.

The second approach is called software refactoring at the package level using clustering with variable number of packages. When tested on an industrial system, this approach showed a high efficiency in auto-packaging of the classes in such way as to balance intra-package cohesion with inter-package coupling. During the experiments on the last approach, we used three traditional clustering algorithms SLINK, CLINK, WPGMA and our proposed algorithm A-KNN. The performances of A-KNN and SLINK are comparable, and both perform better than the other algorithms. In addition, A-KNN has better computational complexity than all other algorithms (SLINK, CLINK and WPGMA).

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

8.1. Introduction

In this chapter, we present our conclusions and we suggest directions for future research in this field. This chapter is organized as follows. Section 2 presents the summary and our conclusions. Section 3 states the possible future work.

8.2. Conclusions and Summary

Refactoring requires a lot of effort from the software designer to make decisions. In this thesis, we investigated software refactoring by using pattern recognition techniques. These techniques assist the software designer to decide on how to refactor software so as to balance coupling and cohesion. This thesis presents five approaches to refactoring based on pattern recognition techniques. These approaches cover three levels: function, class, and package. Different clustering techniques were used as engines for these

approaches, and different similarity measures were proposed and used as controllers for these clustering algorithms. We adapted and used two main types of clustering techniques: with a fixed number of clusters, and a variable number of clusters. Different comparisons were conducted to test the performance of these techniques. This thesis introduces a clustering algorithm called Adaptive K-Nearest Neighbor (A-KNN), and we conducted a set of experiments to test the efficiency of this algorithm for pattern recognition.

At the function level of refactoring, we investigated the selection of entities and attributes, software metrics, similarity measures, resemblance coefficient experiments, hierarchical agglomerative algorithms, and the application of the approach to a source code. By doing a comparative study, we chose the best of three traditional clustering techniques (SLINK, CLINK and WPGMA), using the most suitable similarity measure weights. WPGMA outperformed SLINK and CLINK and the weight ratio of 8:3 was the best for the previous three algorithms. Then we compared our introduced algorithm A-KNN with the previous three algorithms, and our results showed that A-KNN requires less computation and is competitive with the others and in some steps better than all of them.

At the class level, we proposed two new approaches, which both use clustering algorithms. The first approach has a fixed number of classes and its goal was to correct the structure-errors in the current system classes. This was done by providing refactoring advice to move methods from one class to others. Thus, a fixed number of cluster centers is the number of the current classes in the system. Each of the methods in the system will

be assigned to one of these classes according to its similarity with the cluster center. The similarity measure between the method and a class is taken as the number of attributes that the method uses from that class. We explored the different settings of this approach and its controllers. The performance of the presented approach was tested by a set of experiments, conducted on a test source code and an industrial system. The approach was proven to give helpful advice to software designers.

The second approach has a variable number of classes, and it aims to provide a new design for the system classes. Therefore, the classes' methods were used as entities. These entities are clustered according to their similarity with each other. The similarity between two methods depends on how many shared classes' data-members they use and how many times those data-members are accessed by the entities. We tested the approach by using three clustering algorithms SLINK, CLINK, and WPGMA. The experiments showed that the approach is helpful for software designers. Then we tested the performance of our proposed algorithm A-KNN as a suitable engine for that approach. A-KNN continued to show its superiority over the traditional clustering algorithms SLINK, CLINK, and WPGMA.

At the package level, we proposed two approaches. The first has a fixed number of packages. We investigated the similarity measure between the classes and packages. The number of class instances inside a package proved to be a suitable similarity measure between the class and the package. Thus, the classes are assigned to packages according to their similarity with those packages. After proposing the adaptive clustering approach

to distribute the entities on the fixed number of classes, we applied our approach on a made-up code. The results of this proved that the approach is useful to balance coupling with cohesion. Later, we applied the proposed approach on an industrial system, and again it proved to be useful in balancing coupling and cohesion.

The second approach has a variable number of packages, and its purpose is to provide a way for "auto-packaging" of the classes guided by the intra-package cohesion and inter-package coupling using clustering techniques. The entities to be clustered are the classes. The similarity between two classes depends on how many shared methods they use and how many times the classes access these methods. We tested the approach on an industrial system, and it showed a high efficiency in auto-packaging of the classes. During the experiments on the last approach, we used three traditional clustering algorithms SLINK, CLINK, WPGMA and our proposed algorithm A-KNN. A-KNN and SLINK outperformed the other algorithms. However, A-KNN requires less computational complexity than SLINK.

This thesis presented a clustering algorithm which we called Adaptive K-Nearest Neighbor (A-KNN). It depends mainly on the K-Nearest Neighbor algorithm, and it adapts the classification mechanism of K-Nearest Neighbor to perform clustering on data elements. A-KNN considers each element as a single cluster. Then, it works in a similar way to SLINK. However, A-KNN labels a cluster by the label of K-Nearest cluster to it and not by the nearest cluster as in SLINK. A-KNN works in a recursive manner until all the clusters are merged into one at the top level of the hierarchy. Using this algorithm,

three experiments were conducted with different data sets, different controllers, different settings and different objectives. In each experiment, we compared the performance of A-KNN with that of SLINK, CLINK and WPGMA on the same data set under the same conditions. In all our experiments, A-KNN showed either competitive or better performance than all of the other algorithms. Since A-KNN calculates the similarity between single elements instead of between two groups (as in traditional clustering algorithms), A-KNN requires less computation than all of them.

8.3. Future work

This work can be extended by investigating other similarity measures between the entities. The approaches with a fixed number of clusters could use different similarity measures between the methods and classes at the class level, and different similarity measures between the classes and packages at the package level. Similarly, the approaches with a variable number of clusters could use different similarity measures between lines of code at the method level, between the methods at the class level and between the classes at the package level.

A-KNN may also used as a clustering algorithm in other fields of research. In our experiments, this algorithm showed its superiority over the traditional clustering algorithms. However, our experiments were related to software engineering, whereas future research can extend A-KNN to different fields.

APPENDIX A: The source code before refactoring

The First Package

```
package package1;
import package2.*;
public class A {
public  int a_attr1;
public int a_attr2;
public double a_attr3;
public double a_attr4;
 F f1=new F();
  public A() {
  }
  int method0(){
    return this.a_attr1+3*f1.f_attr1;
  }
  double method1(){
     return this.a_attr1+this.a_attr2+this.a_attr3+ this.a_attr4;
  }
  public void method2(){
    this.a_attr4=this.a_attr1+this.a_attr2+this.a_attr3;
  }
}
```

```
package package1;
import package2.*;
public class B {
public  int b_attr1;
public  int b_attr2;
public  double b_attr3;
public  double b_attr4;
  F f1=new F();

  public B() {
  }
  public double method3(){
    return (this.b_attr1+this.b_attr2+this.b_attr3)/3;
  }
  public void method4(){
    this.b_attr4=(this.b_attr1+this.b_attr2+this.b_attr3)/3;
  }
  public double method5(){
    return this.b_attr4*f1.f_attr4;
  }
}
```

```
package package1;
import package2.*;
public class C {
public  int c_attr1;
public  int c_attr2;
public  double c_attr3;
public  double c_attr4;
 F f1=new F();
 A a1=new A();
  public C() {
  }
  public void method6(){
    this.c_attr3=f1.f_attr1*0.4;
    this.c_attr4=(this.c_attr2+this.c_attr3+this.c_attr4)*a1.a_attr1;
  }

}
```

```
package package1;
import package2.*;
```

```
import package3.*;
public class D {
public  int d_attr1;
public  int d_attr2;
public  double d_attr3;
public  double d_attr4;
  K k1=new K();
 F f1=new F();
 L l1=new L();
  public D() {
  }
  public void method7(){
   this.d_attr1=5;
   this.d_attr1=(int) l1.l_attr3;
  }
  public double methdod8(){
  return k1.k_attr1*f1.f_attr1*l1.l_attr2;
  }
}
```

```
package package1;
import package2.*;
public class E {
public  int e_attr1;
public  int e_attr2;
public  double e_attr3;
public  double e_attr4;
  F f1=new F();
  public E() {
  }
  public double method9(){
  return    this.e_attr3=f1.f_attr3+f1.f_attr4;
  }
  public void method10(){
    this.e_attr4=f1.f_attr4* 0.5;
  }
}
```

The Second Package

```
package package2;
import package1.*;
public class F {
public  int f_attr1;
public  int f_attr2;
public  double f_attr3;
public  double f_attr4;
  G g1=new G();
  A a1=new A();
  public F() {
  }
  public void method11(){
   this.f_attr1=g1.g_attr1+a1.a_attr1;
  }
  public void method12(){
    this.f_attr2=3;
  }
}
```

```
package package2;
import package3.*;
public class G {
public  int g_attr1;
public  int g_attr2;
public  double g_attr3;
public  double g_attr4;
 F f1=new F();
```

```
 K k1=new K();
 public G() {
  }
  public double method13(){
    return (this.g_attr3+this.g_attr4)/2;
  }
  public double method14(){
    return this.g_attr4 / 4;
  }
}
```

```
package package2;
import package3.*;
public class H {
public  int h_attr1;
public  int h_attr2;
public  double h_attr3;
public  double h_attr4;
K k1=new K();
  I i1=new I();
  public H() {
  }
  public void method15(){
    this.h_attr1=k1.k_attr1+i1.i_attr2;
  }
  public void method16(){
    this.h_attr4=this.h_attr2+this.h_attr4;
  }
}
```

```
package package2;
import package3.*;
public class I {
public  int i_attr1;
public  int i_attr2;
public  double i_attr3;
public  double i_attr4;
J j1=new J();
  K k1=new K();
  public I() {
  }
  public int method17(){
    return this.i_attr1*j1.j_attr1;
  }
  public void method18(){
    this.i_attr1=2;
    this.i_attr2=2;
    this.i_attr3=5.5;
    this.i_attr4=8.4;
  }
}
```

```
package package2;
public class J {
public  int j_attr1;
public  int j_attr2;
public  double j_attr3;
public  double j_attr4;
  F f1=new F();
  I i1=new I();
  public J() {
  }
  public int method19(){
    return this.j_attr1+this.j_attr2;

  }
  public double method20(){
    this.j_attr3=f1.f_attr3*5;
    return this.j_attr3+this.j_attr4;
```

```
  }
}
```

The Third Package

```
package package3;
import package1.*;
import package2.*;
public class K {
public  int k_attr1;
public  int k_attr2;
public  double k_attr3;
public  double k_attr4;
  G g1=new G();
  D d1=new D();
  public K() {
  }
  public void method21(){
    this.k_attr1=2;
    this.k_attr2=2;
    this.k_attr1=3;
    this.k_attr4=3;
  }
  public void method22(){
    this.k_attr1=g1.g_attr1;
    this.k_attr2=g1.g_attr2;
    this.k_attr3=d1.d_attr3;
    this.k_attr4=d1.d_attr4;

  }
}
```

```
package package3;
import package1.*;
public class L {
public  int l_attr1;
public  int l_attr2;
public  double l_attr3;
public  double l_attr4;
 D d1=new D();
 O o1=new O();
  public L() {
  }
  public double method23(){
    return this.l_attr4+d1.d_attr4+o1.o_attr4;
  }
  public double method24(){
    return this.l_attr4+this.l_attr3;
  }
}
```

```
package package3;
public class M {
public  int m_attr1;
public  int m_attr2;
public  double m_attr3;
public  double m_attr4;
  N n1=new N();
  O o1=new O();
  public M() {
  }
  public int method25(){
    return this.m_attr1+this.m_attr2;
  }
  public int method26(){
    return this.m_attr1+n1.n_attr1+o1.o_attr1;
  }
```

```
}
package package3;
import package2.*;
public class N {
public  int n_attr1;
public  int n_attr2;
public  double n_attr3;
public  double n_attr4;
  K k1=new K();
  G g1=new G();
  public N() {
  }
  public void method27(){
    this.n_attr1=k1.k_attr1+g1.g_attr1;
  }
  public void method28(){
    this.n_attr3=k1.k_attr3;
    this.n_attr4=k1.k_attr4;
  }
}
package package3;
import package1.*;
import package2.*;
public class O {
public  int o_attr1;
public  int o_attr2;
public  double o_attr3;
public  double o_attr4;
  D d1=new D();
  G g1=new G();
  public O() {
  }
  public void method29(){
    this.o_attr1=d1.d_attr1;
    this.o_attr2=g1.g_attr2*2;
  }
  public double method30(){
    return (this.o_attr1+this.o_attr2)/(this.o_attr3+this.o_attr4);
  }
}
```

APPENDIX B: The source code after refactoring

The First Package

```
package package1;
public class A {
public  int a_attr1;
public int a_attr2;
public double a_attr3;
public double a_attr4;
 F f1=new F();
  public A() {
  }
  int method0(){
    return this.a_attr1+3*f1.f_attr1;
  }
  double method1(){
     return this.a_attr1+this.a_attr2+this.a_attr3+ this.a_attr4;
  }
  public void method2(){
    this.a_attr4=this.a_attr1+this.a_attr2+this.a_attr3;
  }
}
```

```
package package1;
public class B {
public  int b_attr1;
public  int b_attr2;
public  double b_attr3;
public  double b_attr4;
  F f1=new F();

  public B() {
  }
  public double method3(){
    return (this.b_attr1+this.b_attr2+this.b_attr3)/3;
  }
  public void method4(){
    this.b_attr4=(this.b_attr1+this.b_attr2+this.b_attr3)/3;
  }
  public double method5(){
    return this.b_attr4*f1.f_attr4;
  }
}
```

```
package package1;
public class C {
public  int c_attr1;
public  int c_attr2;
public  double c_attr3;
public  double c_attr4;
 F f1=new F();
 A a1=new A();
  public C() {
  }
  public void method6(){
    this.c_attr3=f1.f_attr1*0.4;
    this.c_attr4=(this.c_attr2+this.c_attr3+this.c_attr4)*a1.a_attr1;
  }
}
```

```
package package1;
public class E {
public  int e_attr1;
public  int e_attr2;
public  double e_attr3;
public  double e_attr4;
```

```
 F f1=new F();
 public E() {
 }
 public double method9(){
 return    this.e_attr3=f1.f_attr3+f1.f_attr4;
 }
 public void method10(){
  this.e_attr4=f1.f_attr4* 0.5;
 }
}
```

```
package package1;
import package3.*;
public class F {
public  int f_attr1;
public  int f_attr2;
public  double f_attr3;
public  double f_attr4;
 G g1=new G();
 A a1=new A();
 public F() {
 }
 public void method11(){
  this.f_attr1=g1.g_attr1+a1.a_attr1;
 }
 public void method12(){
  this.f_attr2=3;
 }
}
```

```
package package1;
import package3.*;
public class L {
public  int l_attr1;
public  int l_attr2;
public  double l_attr3;
public  double l_attr4;
 D d1=new D();
 O o1=new O();
 public L() {
 }
 public double method23(){
  return this.l_attr4+d1.d_attr4+o1.o_attr4;
 }
 public double method24(){
  return this.l_attr4+this.l_attr3;
 }
}
```

The second Package

```
package package2;
public class H {
public  int h_attr1;
public  int h_attr2;
public  double h_attr3;
public  double h_attr4;
K k1=new K();
 I i1=new I();
 public H() {
 }
 public void method15(){
  this.h_attr1=k1.k_attr1+i1.i_attr2;
 }
 public void method16(){
  this.h_attr4=this.h_attr2+this.h_attr4;
 }
```

```
}
package package2;
public class I {
public  int i_attr1;
public  int i_attr2;
public  double i_attr3;
public  double i_attr4;
J j1=new J();
  K k1=new K();
  public I() {
  }
  public int method17(){
    return this.i_attr1*j1.j_attr1;
  }
  public void method18(){
    this.i_attr1=2;
    this.i_attr2=2;
    this.i_attr3=5.5;
    this.i_attr4=8.4;
  }
}
package package2;
import package1.*;
public class J {
public  int j_attr1;
public  int j_attr2;
public  double j_attr3;
public  double j_attr4;
  F f1=new F();
  I i1=new I();
  public J() {
  }
  public int method19(){
    return this.j_attr1+this.j_attr2;

  }
  public double method20(){
    this.j_attr3=f1.f_attr3*5;
    return this.j_attr3+this.j_attr4;
  }
}
package package2;
import package3.*;
public class K {
public  int k_attr1;
public  int k_attr2;
public  double k_attr3;
public  double k_attr4;
  G g1=new G();
  D d1=new D();
  public K() {
  }
  public void method21(){
    this.k_attr1=2;
    this.k_attr2=2;
    this.k_attr1=3;
    this.k_attr4=3;
  }
  public void method22(){
    this.k_attr1=g1.g_attr1;
    this.k_attr2=g1.g_attr2;
    this.k_attr3=d1.d_attr3;
    this.k_attr4=d1.d_attr4;

  }
}
```

The Third Package

```
package package3;
import package2.*;
import package1.*;
public class D {
public  int d_attr1;
public  int d_attr2;
public  double d_attr3;
public  double d_attr4;
  K k1=new K();
 F f1=new F();
 L l1=new L();
  public D() {
  }
  public void method7(){
   this.d_attr1=5;
   this.d_attr1=(int) l1.l_attr3;
  }
  public double methdod8(){
  return k1.k_attr1*f1.f_attr1*l1.l_attr2;
  }
}
```

```
package package3;
import package1.*;
import package2.*;
public class G {
public  int g_attr1;
public  int g_attr2;
public  double g_attr3;
public  double g_attr4;
 F f1=new F();
 K k1=new K();
 public G() {
  }
  public double method13(){
    return (this.g_attr3+this.g_attr4)/2;
  }
  public double method14(){
    return this.g_attr4 / 4;
  }
}
```

```
package package3;
public class M {
public  int m_attr1;
public  int m_attr2;
public  double m_attr3;
public  double m_attr4;
  N n1=new N();
  O o1=new O();
  public M() {
  }
  public int method25(){
    return this.m_attr1+this.m_attr2;
  }
  public int method26(){
    return this.m_attr1+n1.n_attr1+o1.o_attr1;
  }
}
```

```
package package3;
import package2.*;
public class N {
public  int n_attr1;
public  int n_attr2;
public  double n_attr3;
```

```
public  double n_attr4;
  K k1=new K();
  G g1=new G();
  public N() {
  }
  public void method27(){
    this.n_attr1=k1.k_attr1+g1.g_attr1;
  }
  public void method28(){
    this.n_attr3=k1.k_attr3;
    this.n_attr4=k1.k_attr4;
  }
}
```

```
package package3;
public class O {
public  int o_attr1;
public  int o_attr2;
public  double o_attr3;
public  double o_attr4;
  D d1=new D();
  G g1=new G();
  public O() {
  }
  public void method29(){
    this.o_attr1=d1.d_attr1;
    this.o_attr2=g1.g_attr2*2;
  }
  public double method30(){
    return (this.o_attr1+this.o_attr2)/(this.o_attr3+this.o_attr4);
  }
}
```

**********END**********

REFERENCES

[1] D. M. Coleman, D. Ash, B. Lowther, and P. W. Oman, "Using metrics to evaluate software system maintainability," IEEE Computer, vol. 27, no. 8, pp. 44–49, 1994.

[2] T. Guimaraes, "Managing application program maintenance expenditure," Comm. ACM, vol. 26, no. 10, pp. 739–746, 1983.

[3] B. P. Lientz and E. B. Swanson, "Software maintenance management: a study of the maintenance of computer application software in 487 data processing organizations," Addison-Wesley,1980.

[4] R. S. Arnold, "An introduction to software restructuring," in Tutorial on Software Restructuring, IEEE, 1986.

[5] W. G. Griswold and D. Notkin, "Automated assistance for program restructuring," Trans. Software Engineering and Methodology, ACM, vol. 2, no. 3, pp. 228–269, 1993.

[6] W. F. Opdyke, "Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks," Ph.D. thesis, University of Illinois at Urbana-Champaign, 1992.

[7] M. Fowler, "Refactoring: Improving the Design of Existing Programs," Addison-Wesley, 1999.

[8] E. J. Chikofsky and J. H. Cross, "Reverse engineering and design recovery: A taxonomy," IEEE Software, vol. 7, no. 1, pp. 13–17, 1990.

[9] R. S. Pressman, "Software Engineering A practitioner's Approach," 6th Edition, 2005.

[10] E. N. Fenton, S. L. Pfleeger, "Software Metrics – A Rigorous & Practical Approach," PWS Publishing company, Boston, 1997.

[11] C. Larman, "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process," Prentice-Hall, Inc. US, 2002.

[12] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," IEEE Transactions on Software Engineering , Vol. 20, No. 6, 1994.

[13] W. Li, "Another metric suite for object-oriented programming," The Journal of Systems and Software, pp. 155-162, Elsevier, 1998.

[14] W. Li, “Applying Software Maintenance Metrics in the Object-Oriented Software Development Life Cycle,” Ph.D. Dissertation, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 1992.

[15] D. B. Roberts, ” Practical Analysis for Refactoring,” Ph.D thesis, 1999.

[16] R. O. Duda, P. E. Hart, D. G. Strok, “Pattern Classification,” Second Edition.

[17] J. González, I. Rojas, H. Pomares, J. Ortega, Alberto Prieto, “A New Clustering Technique for Function Approximation,” IEEE transactions on neural networks, Vol. 13, No. 1, 2002.

[18] M.R Anderberg, “Cluster Analysis for Applications,” Academic Press, NY, 1973.

[19] B. Everitt, “Cluster Analysis,” Heinemann Educational Books, Ltd., London, 1980.

[20] H.C. Romesburg, “Cluster Analysis for Researchers,” Krieger, Malabar, FL, 1990.

[21] A.K. Kin, R.P.W. Duin, and J. Mao, “Statistical Pattem Recognition: A Review,” IEEE Trans. Pattern Analysis Machine Intelligence, vol. 22, pp. 4-37, 2000.

[22] L. Kaufman and P.J. Rousseeuw, “Finding Groups in Data: An Introduction to Cluster Analysis,” Wiley, 1990.

[23] U. Wattanachon and C. Lursinsap, “Agglomerative Hierarchical Clustering for Nonlinear Data Analysis,” IEEE International Conference on Systems, Man and Cybernetics, 2004.

[24] R. S. Arnold, “Software restructuring,” *Proc. IEEE* **77** (1989) (4), pp. 607–617.

[25] A. Lakhotia and J.C. Deprez, “Restructuring functions with low cohesion,” in: Proc. Work. Conf. Reverse Eng., pp. 36–46, 1999.

[26] H. S. Kim and Y.R. Kwon, “Restructuring programs through program slicing,” *Int. J. Softw. Eng. Knowl. Eng.* **4** , pp. 349–368, 1994.

[27] B.-K. Kang and J.M. Beiman, “Using design abstractions to visualize, quantify, and restructure software,” *J. Syst. Softw.* **42**, pp. 175–187, 1998.

[28] B.-K. Kang and J.M. Beiman, “A quantitative framework for software restructuring,”. J. Softw. Maint.: Res. Pract. 11, pp. 245–284. 1999.

[29] A. Lakhotia and J.C. Deprez, “Restructuring programs by Tucking statements into functions,” J. Inform. Softw. Technol. 40 (11–12), pp. 677–689, 1998.

[30] C.-H Lung and M. Zaman, “Using clustering technique to restructure programs,” in: Proceedings of the International Conference on Software Engineering Research and Practice, 853–858, 2004.

[31] C.-H. Lung, M. Zaman and A. Nandi, “Applications of clustering techniques to software partitioning, recovery and restructuring,” *J. Syst. Softw.* **73** (2), pp. 227–244, 2004.

[32] N. Anquetil and T.C. Lethbridge, “Comparative study of clustering algorithms and abstract representations for software remodularisation”, IEE Proc. Softw. 150 (3), pp. 185–201, 2003.

[33] N. Anquetil, C. Fourrier, and T. Lethbridge, “Experiments with hierarchical clustering algorithms as software remodularization methods”, In: Proc. Work. Conf. Reverse Eng., pp. 235–255, 1999.

[34] J. M. Bieman and B.-K. Kang, “Measuring design-level cohesion,” *IEEE Trans. Softw. Eng.* **24** , pp. 111–124, 1998.

[35] J.M. Bieman, “Measuring functional cohesion,” *IEEE Trans. Softw. Eng.* **20** (1994) (8), pp. 644–657.

[36] E.J. Chikofsky and J.H. Cross II, “Reverse engineering and design recovery: a taxonomy,” *IEEE Softw.* **7** (1), pp. 13–17, 1990.

[37] A.C. Choi and W. Scacchi, “Extracting and restructuring the design of large software systems,” *IEEE Softw.* **7** , pp. 66–71, 1990.

[38] W.C. Chu, S. Patel, “Software restructuring by enforcing localization and information hiding,” In: Proc. Conf. Softw. Maint., pp. 165–172, 1992.

[39] D. Hutchens and V.R. Basili, “System structure analysis: clustering with data bindings,” *IEEE Trans. Softw. Eng.* 11, pp. 749–757, 1985.

[40] A. Lakhotia, J.C. Deprez, “Restructuring functions with low cohesion,” in: Proc. Work. Conf. Reverse Eng., pp. 36–46, 1999.

[41] A. Lakhotia, “Rule-based approach to computing module cohesion,” in: Proceedings of the 15th International Conference on Software Engineering, pp. 35–44, 1993.

[42] A. Lakhotia, “A unified framework for expressing software subsystem classification techniques,” J. Syst. Softw. 36, pp. 211–231, 1997.

[43] C.-H. Lung, “Software architecture recovery and restructuring through clustering techniques,” in: Proceedings of the Third International Workshop on Software Architecture, pp. 101–104, 1998.

[44] S. Mancoridis, B.S Mitchell, C. Rorres, Y. Chen, E.R. Gansner, “Using automatic clustering to produce high-level system organizations of source code,” in: Proceedings of the Sixth International Workshop on Program Comprehension, pp. 45–52, 1998.

[45] S. Mancoridis, B. Mitchell, Y. Chen, E. Gansner, “Bunch: A clustering tool for the recovery and maintenance of software system organizations of source code,” in: Proceedings of the International Workshop on Program Comprehension, 1999.

[46] O. Maqbool, H.A Babri, “The weighted combined algorithm: a linkage algorithm for software clustering,” in: Proceedings of the Eigth Euromicro Working Conference on Software Maintenance and Reengineering, pp. 15–24, 2004.

[47] B.S. Mitchell, S. Mancoridis, “Comparing the decompositions produced by software clustering algorithm using similarity measurements,” in: Proceedings of International Conference of Software Maintenance, 2001.

[48] H.A. Müller, M.A. Orgun, S.R. Tilley and J.S. Uhl, “A reverse engineering approach to subsystem structure identification,” *J. Softw. Maint.: Res. Pract.* **5** , pp. 181–204, 1993

[49] H.A. Müller, K. Wong and S.R. Tilley, “Understanding Software Systems using Reverse Engineering Technology,” Object-Oriented Technology for Database and Software Systems, World Scientific, pp. 240–252, 1995.

[50] R.W. Schwanke, “An intelligent tool for re-engineering software modularity,” in: Proceedings of the 13th International Conference on Software Engineering, pp. 83–92, 1991.

[51] V. Tzerpos, R.C Holt, “Software botryology automatic clustering of software systems”, in: Proceedings of the 20th Annual International Conference of the IEEE vol. 3, pp. 811–818, 1998.

[52] Z. Wen, V. Tzerpos, “An effectiveness measure for software clustering algorithms,” in: Proceedings of the 12th International Workshop on Program Comprehension, pp. 194–203, 2004.

[53] T.A. Wiggerts, “Using clustering algorithms in legacy systems modularization,” in: Proceedings of the Fourth Working Conference on Reverse Engineering, pp. 33–43, 1997.

[54] H.C. Romesburg, “Cluster Analysis for Researchers,” Krieger Publishing Company, Malabar,” FL 1990.

[55] J.-F. Girard, R. Koschke and R. Schied, “A metric-based approach to detect abstract data types and state encapsulations,” *Autom. Softw. Eng.* **6** , pp. 357–386, 1999.

[56] C-H. Lung, X. Xu, M. Zaman, A. Srinivasan, “Program restructuring using clustering techniques,” The Journal of Systems and Software, pp. 1261–1279, 2006.

[57] H. Dhama, “Quantitative models of cohesion and coupling in software,” J. Syst. Softw. (29), 65–74, 1995.

[58] A. Vacondio, “PDF Split and Merge,” Source Forge, Sep. 22, 2008. [Online]. Available: http://sourceforge.net/projects/pdfsam/. [Accessed: Oct. 1, 2008].

[59] S. Rugaber, K. Stirewalt, and L. Wills, Understanding interleaved code, Automated Software Engineering, 3(1-2):47–76, June 1996.

[60] R. S. Pressman. Software Engineering: A Practitioner’s Approach. McGraw Hill, New York, NY, third edition, 1992.

[61] E. Fix and J. Hodges, “Discriminatory Analysis: Nonparametric Discrimination: Consistency Properties,” 1951.

[62] Consept, “CSGestionnaire,” Source Forge, Jun. 13, 2007. [Online]. Available: http://sourceforge.net/projects/csgestionnaire/ [Accessed: Oct. 15, 2008].

[63] E.Blanz, “JLOC,” Source Forge, Nov. 05, 2007. [Online]. Available: http://sourceforge.net/projects/jloc/ [Accessed: Oct. 19, 2008].

[64] S. CK, S. Shrestha, Y. Gurung, “Front End For MySQL,” Source Forge, May. 06, 2008. [Online]. Available: http://sourceforge.net/projects/ frontend4mysql / [Accessed: Nov. 16, 2008].

[65] M.Fabio, “Trama,” Source Forge, Sep. 15, 2008. [Online]. Available: http://sourceforge.net/projects/trama/ [Accessed: Nov. 22, 2008].

Vita

- Al-Khalid, Abdulaziz Muhammad
- Born in Homs, Syria on April 19, 1982.
- Completed Bachelor of Science in Informatics Engineering with major in Software Engineering and Information Systems from Albaath University, Homs, Syria, in August 2005.
- Completed MS in Computer Science from King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia, in Jan, 2009.
- Email: khalidaziz@kfupm.edu.sa